HZ BOOKS
华章IT

物联网核心技术丛书

IoT开发实战

CoAP卷

徐 凯 编著

机械工业出版社
China Machine Press

作者简介

徐 凯

嵌入式软件工程师，现就职于美的集团洗衣机事业部。擅长嵌入式Web系统和6LoWPAN无线传感网应用，精通物联网应用层协议CoAP和物联网操作系统Contiki。

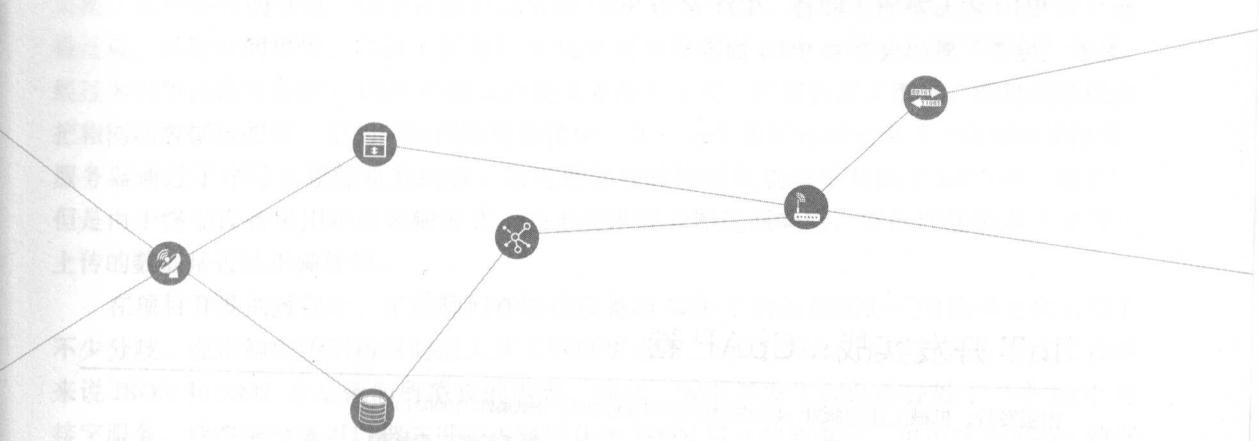
物联网核心技术丛书

CoAP in Action

IoT开发实战

CoAP卷

徐 凯 编著



机械工业出版社
China Machine Press

图书在版编目（CIP）数据

IoT 开发实战：CoAP 卷 / 徐凯编著 . —北京：机械工业出版社，2017.9
（物联网核心技术丛书）

ISBN 978-7-111-57780-5

I. I… II. 徐… III. ①互联网络－应用 ②智能技术－应用 IV. ① TP393.4 ② TP18

中国版本图书馆 CIP 数据核字（2017）第 209172 号

IoT 开发实战：CoAP 卷

出版发行：机械工业出版社（北京市西城区百万庄大街 22 号 邮政编码：100037）

责任编辑：佘 洁　　责任校对：李秋荣

印　　刷：北京诚信伟业印刷有限公司　　版　　次：2017 年 9 月第 1 版第 1 次印刷

开　　本：186mm×240mm 1/16　　印　　张：16

书　　号：ISBN 978-7-111-57780-5　　定　　价：59.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换

客服热线：（010）88379426 88361066　　投稿热线：（010）88379604

购书热线：（010）68326294 88379649 68995259　　读者信箱：hzit@hzbook.com

Preface 前　言

为何写作本书

几年前我作为一名嵌入式工程师参与了一个关于低功耗车载终端的研发项目，该低功耗车载终端中包含一个 GPS 模块和一个 GPRS(2G) 模块，工程师们希望通过最少的能量消耗把终端的 GPS 坐标上传至 Web 服务器中。虽然需求直截了当，但是在开发的过程中却出现了各种各样的分歧。例如传输协议采用 UDP 还是 TCP，有的工程师认为 UDP 没有连接过程，传输时间更短，有的工程师认为 TCP 更加可靠而 UDP 也许会出现“丢包”现象。经过多次争论最终选择了 UDP 传输加自定义重传的方式。所谓自定义重传，就是车载终端把相同的数据包按照一定的时间间隔连续传输三次，每个数据包都包含一个递增的子序号，服务器通过子序号来剔除重复内容。通过增加这种容错机制似乎解决了 UDP 的“缺陷”，但是由于终端设备采用单向传输方式，并不要求服务器返回响应，所以终端根本不知道它上传的数据是否被正确处理。

在项目开发的过程中，工程师们在终端设备与 Web 平台的衔接方式的选择上也出现了不少分歧。在定制应用层协议时嵌入式工程师更喜欢二进制协议，但对于 Web 开发工程师来说 JSON 和 XML 才是他们所擅长的内容。因此，Web 开发工程师单独做了一个 UDP 套接字服务，使终端设备可以把二进制内容转化为 JSON 格式的数据包，再把这个 JSON 数据包“POST”到一个 HTTP 服务器。此时对于 Web 开发工程师来说，设备其实是在提交表单。

经过工程师们的不断努力，这个低功耗车载终端如期完成。但是项目完成之后我不禁思考：这个项目是不是可以做得更好一点，是不是可以打破嵌入式工程师和 Web 开发工程师的技术鸿沟，是不是有更好的应用协议可以满足项目需求，是不是低功耗终端也可以提交表单？查阅了众多资料之后，我找到了 CoAP。

回想硕士毕业之后我“执着”地成为一名专注于物联网的软件工程师，而我本科和硕士的专业都与机械工程相关。与其他计算机或电子专业不同，机械工程特别强调规范和标准，所以设计过程必须严格遵守规范。虽然表面上这显得异常死板，但是这种规范却大大提高了系统的互换性，节约了开发成本。在这种理念的指导下，我总是先寻找标准解决方

案而不是随时随地准备“造轮子”。CoAP 是一个由 IETF（Internet Engineering Task Force，互联网工程任务组）组织编写的面向低功耗设备的物联网应用层协议，协议编号为 RFC 7252。我非常高兴找到了应用“标准”，而不是又找到了一组“轮子”。

CoAP 有很多优点，而这些优点正好可以解决上文提到的低功耗车载终端所遇到的问题：

1）CoAP 传输层协议采用 UDP，对于终端来说 UDP 的确可以减少一部分能耗。

2）CoAP 采用请求 / 响应工作模式，当终端设备发送 CoAP 请求之后，服务器将返回响应码，终端通过响应码可以判断服务器的处理结果。

3）CoAP 包含重传机制，不用再重新设计重传方法。

4）CoAP 参考了 HTTP 的大量成功经验，如 CoAP 请求方法、CoAP 选项定义和 CoAP 响应码等，所以 Web 开发工程师也可以非常容易地掌握 CoAP。

CoAP 可以帮助低功耗智能终端接入网络，通过这种标准协议也可以降低物联网系统的开发难度，尤其可降低物联网 Web 平台的开发难度。对于应用 CoAP 的终端设备来说，同样会遵守 REST 标准，使用类似的资源描述方法，使用相同的请求方法，应用相同的 JSON 数据包。对于物联网 Web 平台来说，处理一次终端设备的数据上传和处理一次 Ajax 表单提交同样容易。

我个人喜欢阅读技术图书，通过阅读图书可以系统地掌握一门新技术，我也希望本书可以帮助读者熟练掌握 CoAP，并把它应用于物联网系统中。

目标读者

本书适合物联网爱好者、嵌入式工程师和 Web 开发工程师。

- 对于物联网爱好者而言，本书的示例可以让你更快地熟悉物联网系统。本书包括很多与物联网系统相关的基础知识，通过这些基础知识的学习可以加深你对物联网系统的理解。通过本书中的多个动手示例，你可以掌握物联网系统的调试方法。
- 对于嵌入式工程师而言，本书可以帮助你从不同角度了解低功耗设备如何连接网络。通过 CoAP 的学习可以从另一个角度熟悉 HTTP。CoAP 和 HTTP 都是设备连接网络的常见手段。
- 对于 Web 开发工程师而言，可以从另一个角度了解设备如何提交“表单”，通过学习 CoAP 你会发现低功耗终端设备也可以很流畅地接入系统，而不需要做多余的协议转换。

如何阅读本书

本书的主要内容大致分为三部分：

第一部分：第 1～3 章。第 1 章介绍与物联网应用直接相关的各种协议，这些协议包括 IP、6LoWPAN 协议、IEEE 802.15.4 协议、HTTP、MQTT 协议和 CoAP 等；第 2 章介绍与物联网应用相关的开源硬件 Arduino 和树莓派，无论是 Arduino 还是树莓派都是开源硬件领

域的“明星产品”，在这些硬件平台上可以快速实现 CoAP；第 3 章与前面两章不同，该章通过多个示例详细介绍与 CoAP 息息相关的网络协议——IP、UDP、TCP 和 HTTP，掌握这些协议是学习 CoAP 的基础。

第二部分：第 4～8 章。在第 4 章中先通过一个简洁示例让读者对 CoAP 有一个大致的了解，该示例包含 CoAP 客户端和 CoAP 服务器两个部分，CoAP 服务器使用 Arduino UNO 实现，通过一个安装了 Copper 插件的 Firefox 浏览器便可访问该 CoAP 服务器；第 5 章与第 6 章详细分析与 CoAP 相关的 RFC 文档，这两章是掌握 CoAP 的理论基础；第 7 章介绍多种 CoAP 客户端和服务器的实现方法，这些实现方法包括 C 语言、Python、Node.js 和 Java；在实际项目中使用 CoAP 难免出现问题，第 8 章介绍 CoAP 的多种调试方法，通过 Copper 插件和 Wireshark 网络抓包工具可以快速地发现 CoAP 的细节错误。

第三部分：9.10 章。最后两章设计一个微型的物联网系统，试图通过该系统向读者展现物联网系统从设计到实现的整个过程。微型物联网系统包括服务器和设备两部分。服务器部分（第 9 章）包括 Web 前端、后端和数据库部分的实现内容，与其他 Web 系统不同，该系统还包括 CoAP 服务器实现；设备部分（第 10 章）使用一个低功耗受限制设备作为 CoAP 客户端，该设备使用 Contiki 作为嵌入式操作系统，使用 IEEE 802.15.4 这样的无线方式连接网络。

相关资料

本书提供多个基础示例，这些示例代码可以帮助读者深入了解 CoAP。

示例代码仓库：https://github.com/xukai871105/the_beginning_of_coap。

本书还提供一个 CoAP 测试服务器，该测试服务器部署于阿里云，国内用户可以非常方便地访问该服务器。

CoAP 测试服务器：coap://wsncoap.org。

勘误和支持

由于时间和水平方面的限制，书中难免出现错误或者描述不准确的地方，恳请读者批评指正。如果读者在阅读过程中发现问题，可通过个人博客或邮箱与我取得联系。

我的邮箱：xukai19871105@126.com。

我的博客：http://blog.csdn.net/xukai871105。

致谢

感谢机械工业出版社华章公司的编辑，没有你们的鼓励就不会有这本书。感谢我的同事崔红鹏、王耀庭、许静和伊明，感谢你们与我一同讨论 CoAP 的各种细节问题，并把 CoAP 真正应用到实际产品中。最后感谢我的妻子左文娟，在这一年多的时间里始终支持我的写作，是你的鼓励让我最终完成本书。

目　录 *Contents*

第 1 章 Chapter 1

物联网与网络协议

1.1 本章主要内容

CoAP 是受限制应用协议（Constrained Application Protocol）的简称，CoAP 是物联网应用层协议之一。当前市面上有很多物联网应用层协议，它们之间既相互关联也存在明显区别。本章将重点介绍三种物联网应用层协议——CoAP、HTTP 和 MQTT，三种协议具有各自不同的适用场景，在物联网领域均有广泛的应用。

HTTP 和 MQTT 使用 TCP 作为传输层协议，而 CoAP 则使用 UDP 作为传输层协议。无论是 UDP 还是 TCP 均依赖于 IP 技术，IP 技术是现代网络通信的基础。IP 又分为 IPv4 和 IPv6 两个版本，IPv4 已经广为人知而 IPv6 才刚刚投入使用。随着智能设备的发展，越来越多的设备需要接入网络，IPv4 地址枯竭的问题越来越严重。6LoWPAN 技术是一种 IPv6 头压缩技术，通过 6LoWPAN 头压缩技术可以有效地解决智能设备通过 IPv6 接入网络的问题。图 1-1 可以很好地概括本章的具体内容。

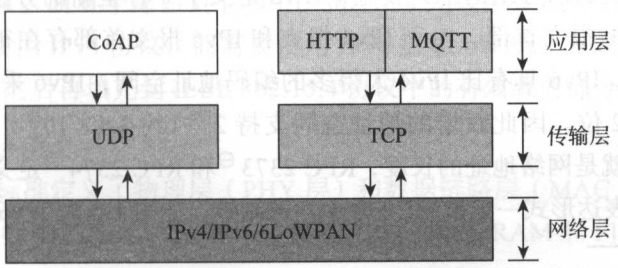


图 1-1　物联网协议概述

1.2 物联网与IP

毫无疑问IP技术是当今互联网应用的基础，同时IP技术也是物联网应用的基础。本章中IP技术分为IPv4、IPv6和6LoWPAN三个部分进行介绍。

1.2.1 IPv4

IPv4即互联网协议版本4，又称互联网通信协议第4版。IPv4为标准化互联网络的核心部分，也是使用最广泛的互联网协议版本，其后继版本为IPv6，直到2011年IANA IPv4 pool地址已经完全用尽时，IPv6仍处在部署的初期。IPv4是一种无连接的协议，此协议会尽最大努力交付分组数据，IPv4不保证任何分组数据均能送达目的地，也不保证所有分组数据均按照正确的顺序无重复地到达。

IPv4使用32位（4字节）地址，因此地址空间中只有4 294 967 296个地址。不过，一些地址为特殊用途所保留，如专用网络（约1800万个地址）和多播地址（约2.7亿个地址），专用网络和多播地址也减少了可在互联网上路由的地址数量。随着地址不断被分配给最终用户，IPv4地址枯竭问题也随之产生。基于NAT（网络地址转换）等地址结构重构的技术显著地降低了IPv4地址枯竭的速度。

毫无疑问IPv4技术是物联网应用的基础。关于终端设备，具有网络连接能力的设备很有可能包含IPv4协议栈，这就意味着该设备可以很容易地访问网络中的任意一个IPv4应用；关于网关设备，对于那些尚没有网络连接能力的设备来说，也可以通过定制的网关设备转发有用内容，这些定制的网关设备往往把非IP数据包转换成IP数据包；关于服务器，绝大多数Web应用、数据库存储应用和搜索服务器均依赖于IPv4技术。对于终端设备、网关设备和服务器而言，当前IPv4技术支撑着整个物联网应用。

但是IPv4技术也不是万能的，随着物联网终端设备的爆发，IPv4地址枯竭的问题显得越来越严重，已经被长期依赖的IPv4技术并不一定是物联网应用未来发展的方向。

1.2.2 IPv6

IPv6即互联网通信协议第6版，是互联网协议的最新版本，旨在解决IPv4地址枯竭问题。在因特网中，数据以分组的形式传输。IPv6定义了一种全新的分组格式，目的是为了最小化路由器处理的报文首部。由于IPv4报文和IPv6报文首部存在很大不同，因此这两种协议无法互操作。IPv6具有比IPv4大得多的编码地址空间，IPv6采用了128位的地址，而IPv4使用的是32位。因此新增的地址空间支持2^{128}（约3.4×10^{38}）个地址。从IPv4到IPv6最显著的变化就是网络地址的长度，RFC 2373[⊖]和RFC 2374[⊜]定义的IPv6地址有128位长，IPv6地址的表达形式一般采用32个十六进制数。如图1-2为IPv6 Ready认证标签。

⊖ https://datatracker.ietf.org/doc/rfc2373/。

⊜ https://datatracker.ietf.org/doc/rfc2374/。

IPv6 技术摒除了 IPv4 技术的多数局限，是一种更加进步与优化的互联网协议，它具有以下优势：

1）强大的地址空间：正如上文所述 IPv6 的地址长度为 128 位，而 IPv4 的地址长度仅为 32 位，强大的地址空间可以满足物联网终端设备数量增长的需求。

2）即插即用：IPv6 采用即插即用的机制实现与各种设备的网络连接，相关配置可以自动生成而并不需要向服务器申请。

3）更高的安全性：IPv6 特性描述中要求通过加密有效载荷和通信源认证等方式增强网络的安全性。

4）更加灵活与完善的首部：IPv6 中移除了 IPv4 中并不常用的字段，如分段与校验和等，采用了固定头加可选扩展头的组合方式。

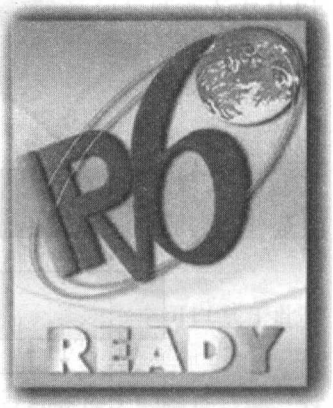


图 1-2　IPv6 Ready 认证标签

1.2.3　6LoWPAN

虽然 IPv6 协议是更为高效和完善的互联网协议，但是对于大多数受限制的物联网设备来说，IPv6 协议依然冗余而复杂。为了让 IPv6 技术能够适用于低功耗受限制物联网设备，6LoWPAN 技术应运而生。

IETF 组织于 2004 年 11 月正式成立了 6LoWPAN 工作组，着手制定基于 IPv6 的低速率无线个域网标准，即 IPv6 over IEEE 802.15.4，该工作组将 IPv6 引入以 IEEE 802.15.4 为标准的无线个域网中。IEEE 802.15.4 是无线个域网技术的典型代表，已经获得了广泛的应用。但 IEEE 802.15.4 标准只规定了物理层和媒体访问控制层两部分，并没有涉及网络层以上规范。

图 1-3 可以很好地说明 6LoWPAN 与 IPv6、IEEE 802.15.4 之间的关系。图 1-3 的左侧部分说明了那些非受限制设备如何与互联网建立连接，大多数 Linux 主机都属于非受限制设备，这些设备往往具备足够内存空间和很好的运算能力，如市面上常见的树莓派。对于树莓派这样的非受限制设备，可以使用 IPv4 层作为网络层协议，使用 IEEE 802.3 作为物理层和链路层协议。而对于那些具有 IEEE 802.15.4 无线连接能力的受限制低功耗设备来说，并不能直接使用 IPv4 协议，而需要使用 IPv6 加 6LoWPAN 方式，把 IPv6 首部经过 6LoWPAN 技术压缩之后再填充到 IEEE 802.15.4 协议中的有效载荷部分。

1. IEEE 802.15.4 简介

IEEE 802.15.4 标准定义了物理层（PHY 层）和数据链路层（MAC 层）。市面上有不少符合 IEEE 802.15.4 标准的 SoC，这些 SoC 虽然资源（内部 RAM 和 Flash）受限，功能较低，但是成本低廉。

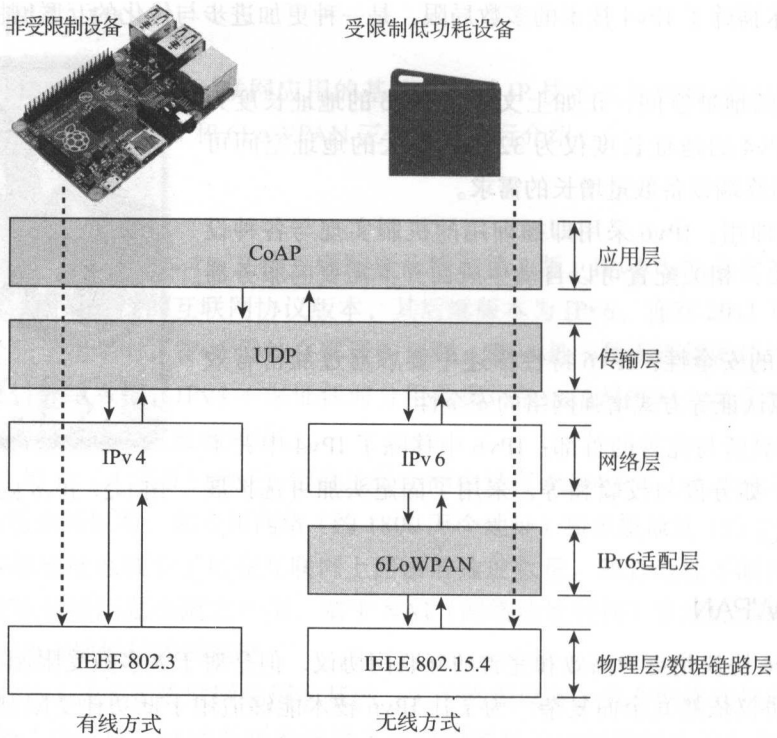


图 1-3 6LoWPAN 适配层作用

IEEE 802.15.4 标准的主要特征如下：

1）低速率：在 2.4GHz 频段最大速度为 250Kbit/s。

2）地址短：支持 16 位短地址。

3）低功耗：可支持电池供电应用。

4）低成本：可适用于资源受限制设备。

5）短距离：节点信号覆盖范围一般为 10～100 米，覆盖范围有限。

6）低复杂度：相比于 IEEE 802.11 和 IEEE 802.15.1，IEEE 802.15.4 相对简单，容易实现。

7）短帧长：IEEE 802.15.4 数据链路层的最大传输单元为 127 字节，只能为输出的数据提供较少的有效空间。

8）多种拓扑结构：IEEE 802.15.4 标准支持点对点和星形网络。

2. 6LoWPAN 简介

因为 IPv6 要求数据链路层支持的最小传输单元为 1280 字节，而 IEEE 802.15.4 链路的最大传输单元仅为 127 字节，所以需要在网络层之下定义一个适配层，负责 IP 数据包的压缩、分片和重组等工作。

6LoWPAN 适配层之下采用 IEEE 802.15.4 规定的物理层和数据链路层，6LoWPAN 适配层之上的网络层采用 IPv6 协议。由于在 IPv6 中数据链路层支持的载荷长度远大于 IEEE 802.15.4 所能提供的最大载荷长度，为了实现 IEEE 802.15.4 与网络层（IPv6）的无缝链接，6LoWPAN 适配层被增加至网络层和 IEEE 802.15.4 之间，6LoWPAN 适配层用来完成头压缩、分片与重组以及网状路由转发等工作。

6LoWPAN 技术具有如下优势：

1）普及性：IP 应用非常广泛，作为下一代互联网核心技术的 IPv6 也在加速其普及的步伐。

2）适用性：IP 网络协议栈架构受到广泛的认可，低速率无线个域网完全可以基于此架构进行简单、有效的开发。

3）更多地址空间：IPv6 应用于低功耗无线个域网的最大亮点就是庞大的地址空间。这恰恰满足了部署大规模、高密度设备的需要。

4）支持无状态自动地址配置：IPv6 中当节点启动时，可以自动读取 MAC 地址，并根据相关规则配置好所需的 IPv6 地址。

5）易接入：低速率无线个域网使用 IPv6 技术，更易于接入其他基于 IP 技术的网络及下一代互联网，使其可以充分利用 IP 网络的技术进行快速发展。

图 1-4 可以很好地说明 6LoWPAN 头压缩技术如何与 IEEE 802.15.4 标准配合工作。IEEE 802.15.4 标准的物理层部分一般包括先导码、同步字和物理层长度指示域和物理层有效载荷等部分；在数据链路层部分又可分为帧控制域、序列号域、地址域、数据链路层有效数据载荷与校验区域等部分；在数据链路层有效载荷部分又可分为 IPv6 压缩头部分和 IPv6 有效负载部分。

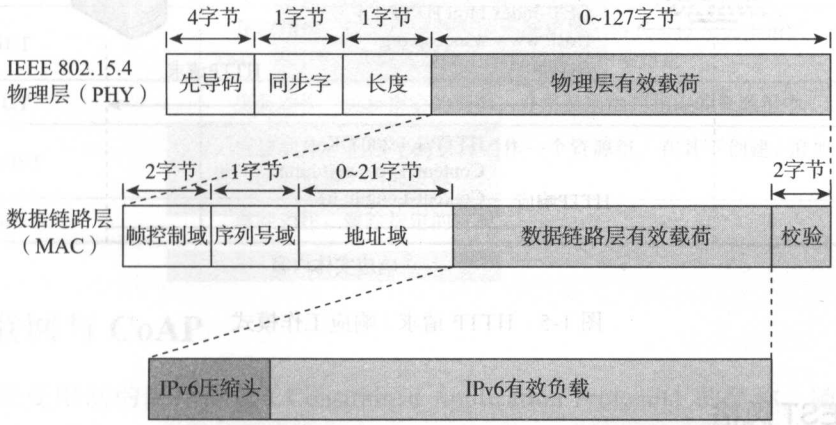


图 1-4　6LoWPAN 头压缩技术与 IEEE 802.15.4 之间的关系

1.3 物联网与 HTTP

在互联网领域 HTTP 是应用最为广泛的应用层协议，在物联网领域 HTTP 也是不可或缺的重要组成部分。HTTP 的成功应用也影响了物联网领域的专用协议，如本书讨论的 CoAP 借鉴了 HTTP 在应用过程中大量的成功经验。熟练掌握 HTTP 对物联网领域的专项内容学习绝对大有帮助。

1.3.1 HTTP

HTTP（超文本传输协议）是互联网上应用最为广泛的一种网络协议。设计 HTTP 最初的目的是为了提供一种发布和接收 HTML 页面的方法。通过 HTTP 或 HTTPS 请求的资源由统一资源标识符（Uniform Resource Identifier，URI）来标识。

HTTP 采用客户端请求 - 服务器响应这样的工作模式。图 1-5 可以很好地描述这种请求 / 响应工作模式。通常客户端使用网页浏览器向服务器上指定端口（HTTP 的默认端口为 80）发起一个 HTTP 请求。服务器上存储着很多资源，如普通文本、HTML 文本、图片或视频文件等。我们称这个应答服务器为源服务器。在用户代理和源服务器中间可能存在多个“中间层”，比如代理服务器、网关或隧道等。HTTP 采用 TCP 作为其传输层协议。通常，由客户端发起一个 HTTP 请求，创建一个到服务器指定端口的 TCP 连接。HTTP 服务器则在那个端口监听客户端的请求，一旦收到请求，服务器将会向客户端返回状态，如“HTTP/1.1 200 OK”，以及具体的响应内容，如文本文件、HTML 文件、图片和视频文件等。

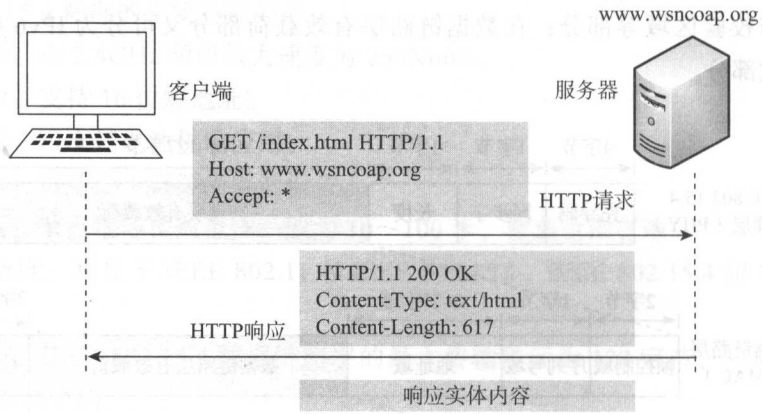


图 1-5 HTTP 请求 / 响应工作模式

1.3.2 REST 风格

REST（具象状态传输）是 Roy Thomas Fielding 博士于 2000 年在他的博士论文“Architectural Styles and the Design of Network-based Software Architectures”中提出来的一种 Web 软

件架构风格。目前在三种主流的 Web 服务实现方案中，REST 模式与复杂的 SOAP 和 XML-RPC 相比更加简洁，越来越多的 Web 服务开始采用 REST 风格设计和实现。REST 是设计风格而不是标准，REST 通常基于使用 HTTP、URI、XML、JSON 和 HTML 这些现有的协议和标准来实现。

REST 风格具有以下特点：

1）资源一般由 URI 来指定。

2）无状态通信。

3）对资源的操作包括创建、获取、修改和删除等，这些操作对应 HTTP 的 GET、POST、PUT 和 DELETE 方法。

4）资源的表现形式可以是 XML、JSON 或 HTML 格式文件。

REST 设计风格确实可以带来一些好处，这些好处使 Web 开发更加简洁且易于实现。REST 风格可以使资源的定义方式更加清晰，Web 服务器中常常保存很多不同类型的资源，这些资源需要通过某种约定俗成的方法加以编号，若遵循 REST 风格那么 www.wsncoap.org 中某个具体的资源可能采用这样的 URI 定义：http://wsncoap.org/resources/15。

REST 风格使得对资源的操作变得更加简洁，若采用 REST 风格可充分利用 HTTP 中已经使用的各种“动词”——GET、PUT、POST 和 DELETE。GET 代表获取、PUT 代表更新、POST 代表创建而 DELETE 代表删除，这就避免了在 HTTP 负载部分还需要创建诸如 Create 或 Update 这样“非标准”的动作。另外，HTTP 中 4 个常用方法也对应数据库中的“增删改查”四大操作。表 1-1 可以很好地说明在 REST 风格指导下如何使用 HTTP 中的 4 个常用方法操作 wsncoap.org/resources/15 资源。

表 1-1　REST 风格资源操作说明

HTTP 方法	操 作 说 明
GET	获取动作：获取指定资源的详细信息，格式可以为文本格式、JSON 格式、XML 格式等。资源的媒体类型可以由客户端指定
PUT	更新动作：更新指定的资源，并将其追加到相应的资源组中
POST	创建动作：把指定的资源当作一个资源组，在其下创建 / 追加一个新的元素，使其隶属于当前资源
DELETE	删除动作：删除指定的资源

1.4　物联网与 CoAP

CoAP 是受限制的应用协议（Constrained Application Protocol）的简称。随着最近几年物联网技术的发展，越来越多的设备接入互联网。据预测未来将有更多的设备需要相互连接，而这些设备的数量将远超人类的总和。在这种大背景下，物联网 IoT 和 M2M 技术应运而生。虽然对人们而言，接入互联网显得非常方便，但是对于那些低功耗受限制设备

而言接入互联网却异常困难。在当前由 PC 和智能手机组成的世界里，信息交换多是通过 TCP 和应用层协议 HTTP 实现的。但是对于低功耗受限制设备而言，实现 TCP 和 HTTP 显然是一个过分而苛刻的要求。为了让低功耗受限制设备可以接入互联网，CoAP 应运而生。CoAP 是一种应用层协议，它运行于 UDP 之上而不是像 HTTP 那样运行于 TCP 之上。CoAP 非常小巧，最小的数据包仅为 4 字节。

1.4.1 CoAP

CoAP 并不能孤立存在，而是 TCP/IP 协议族的一部分。TCP/IP 为人熟知，从字面意思上理解 TCP/IP 指 TCP 与 IP 这两种协议，但实际上 TCP/IP 是一类协议集合的统称，具体来说 TCP/IP 包括 IP 和 ICMP、TCP 和 UDP、TELNET 和 FTP 等。虽然 CoAP 并没有使用 TCP 作为传输层协议，但 CoAP 也属于 TCP/IP 协议族中的一员。CoAP 借鉴了 HTTP 的大量成功经验，CoAP 和 HTTP 一样均使用请求 / 响应工作模式。通常由客户端发送 CoAP 请求，服务器一旦侦听到该请求便会根据请求内容返回响应码和响应内容。图 1-6 可以很好地说明 CoAP 请求 / 响应工作模式的大致流程。虽然 CoAP 和 HTTP 有很多相似之处，但是 CoAP 专门为低功耗受限制设备设计，它比 HTTP 简单很多。

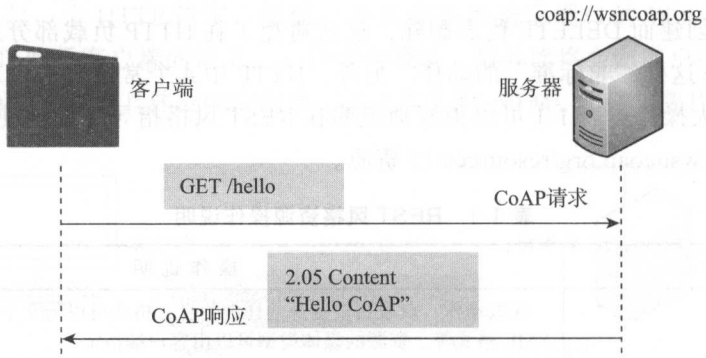


图 1-6 CoAP 请求 / 响应工作模式

1.4.2 RFC 文档汇总

俗话说“没有规矩不成方圆”，这些 TCP/IP 相关协议均由 IETF 组织讨论并制定，IETF 组织是一个坚持开放性和适用性的国际标准化组织，该组织产生的标准化文档被称为 RFC（Request For Comment）文档，所有 RFC 文档完全公开并在互联网上公布。RFC 文档不仅记录了协议规范内容，还包括协议的实现和运用的相关信息。RFC 文档通过编号的方式组织每个协议的标准化请求，如著名的 IP 规范由 RFC 279 规定，而著名的 TCP 规范则由 RFC 793 规定，本书讨论的 CoAP 由 RFC 7252 规定。RFC 编号采用递增方式编号，著名的 IP 和 TCP 的编号仅为 3 位数字，而 CoAP 的编号已经超过 7000。

通过上面的分析不难得出，若需要熟悉并了解 CoAP 可从 RFC 文档入手，通过 CoAP 相关的 RFC 文档可以了解它的“前世今生”。

1. CoAP 核心与扩展协议

CoAP 包括核心协议 RFC 7252 和扩展协议 RFC 7641、RFC 6690 和 RFC 7959 等部分，具体内容见表 1-2。本书后续章节将结合示例详细解释这几份 RFC 文档。

表 1-2　CoAP 核心协议和扩展协议

名　　称	RFC 编号	RFC 文档名称
CoAP 核心协议	RFC 7252	The Constrained Application Protocol（CoAP）
CoAP 观察者模式	RFC 7641	Observing Resources in the Constrained Application Protocol（CoAP）
CoAP 资源描述	RFC 6690	Constrained RESTful Environments（CoRE）Link Format
CoAP 块传输	RFC 7959	Block-Wise Transfers in the Constrained Application Protocol（CoAP）

2. TCP/IP 相关 RFC 文档汇总

CoAP 的应用还依赖于其他 RFC 文档，在这些 RFC 文档的共同支持下才可以组成完整的 CoAP 应用。CoAP 依赖的 RFC 文档见表 1-3。

表 1-3　TCP/IP 相关 RFC 文档汇总

名　　称	RFC 编号	RFC 文档名称
IPv4 说明	RFC 791	Internet Protocol Specification
ICMP 说明	RFC 792	Internet Control Message Protocol
UDP 说明	RFC 768	User Datagram Protocol
TCP 说明	RFC 793	Transmission Control Protocol
DNS 说明	RFC 1034 RFC 1035	Domain Names – Concepts and Facilities Domain Names –Implementation and Specification
DHCP 说明	RFC 2131	Dynamic Host Configuration Protocol
HTTP 1.1 说明	RFC 2616	Hypertext Transfer Protocol
HTTP 2.0 说明	RFC 7540	Hypertext Transfer Protocol Version 2 (HTTP/2)
IPv6 说明	RFC 2460	Internet Protocol, Version 6 (IPv6) Specification
DHCPv6 说明	RFC 3315	Dynamic Host Configuration Protocol for IPv6 (DHCPv6)
NAT64 说明	RFC 6146	Stateful NAT64: Network Address and Protocol Translation from IPv6 Clients to IPv4 Servers
IPv6 over IEEE 802.15.4	RFC 4944	Transmission of IPv6 Packets over IEEE 802.15.4 Networks
IPv6 over BLE	RFC 7668	IPv6 over BLUETOOTH(R) Low Energy
JSON 格式说明	RFC 7159	The JavaScript Object Notation (JSON) Data Interchange Format

（续）

名　称	RFC 编号	RFC 文档名称
CBOR 格式说明	RFC 7049	Concise Binary Object Representation (CBOR)
TLS 1.2 协议说明	RFC 5246	The Transport Layer Security (TLS) Protocol Version 1.2
DTLS 说明	RFC 6347	Datagram Transport Layer Security Version 1.2

1.5 物联网与 MQTT 协议

与 HTTP 和 CoAP 不同，MQTT 协议由 IBM 牵头制定，而 HTTP 与 CoAP 均由 IETF 组织制定。MQTT 协议采用订阅 / 发布模式，这与 HTTP 和 CoAP 的请求 / 响应模式存在明显区别。MQTT 协议虽然不是为物联网应用专门设计的协议，但是在物联网领域依然取得了不俗的成绩。

1.5.1 MQTT 协议

MQ 遥测传输（MQTT[⊖]）是轻量级基于代理的发布 / 订阅模式的消息传输协议，MQTT 协议开放、简单、轻量级且易于实现。该协议的特点有：

1）使用发布 / 订阅消息模式，提供一对多的消息发布，解除应用程序耦合。

2）对负载内容屏蔽的消息传输。

3）使用 TCP/IP 提供网络连接。

4）小型传输，网络传输开销非常小（固定长度的头部是 2 字节），协议交换最小化以降低网络流量。

5）使用 Last Will 和 Testament 特性通知有关各方客户端异常中断的机制。

1.5.2 MQTT 主题

MQTT 协议包含一个主题的概念，MQTT 的主题与 HTTP 中的资源 URI 较为相似。MQTT 协议通过主题对消息进行分类，主题本质上是一个 UTF-8 编码的字符串，可通过斜杠表示多个层级关系。主题还可以使用通配符进行过滤。其中，“+”可以过滤一个层级，而“#”只能出现在主题最后，表示过滤任意级别的层级。

下面是几个常见的 MQTT 主题：

1）building-b/floor-5：表示 B 栋楼第 5 层。

2）+/floor-5：表示任何一栋楼的第 5 层。

3）building-b/#：表示 B 栋楼的所有楼层。

⊖ http://mqtt.org/。

1.5.3　MQTT 服务质量

针对不同的应用场景，MQTT 协议提供三种不同消息发布服务质量：

- QoS=0 “最多一次”：服务质量级别 QoS0 是最快的传输方式，有时称为“触发并忘记”。消息将最多传递一次，或者可能完全不会传递。网络中的传递不会得到确认，并且不会存储消息。如果客户机断开连接或者服务器发生故障，那么消息可能会丢失。以某个时间间隔发送实时数据时，可使用服务质量级别 QoS0。丢失单条消息实际上不会产生很大影响，因为之后很快将发送包含较新数据的另一条消息。在此场景中，使用较高服务质量会带来额外成本，却不会获得任何实际优势。QoS=0 的情况如图 1-7a 所示。
- QoS=1 “至少一次”：使用服务质量级别 QoS1 时消息会始终至少传递一次。如果发布者收到应答之前消息传递失败，那么一条消息可能会传递多次。该消息必须存储在发布者本地，直到发布者收到关于接收者已发布此消息的确认消息为止。QoS=1 的情况如图 1-7b 所示。
- QoS=2 “恰好一次”：服务质量级别 QoS2 是最安全也是最慢的传输方式。消息始终传递恰好一次，并且必须存储在发布者本地，直到发布者收到关于接收者已发布此消息的确认消息为止。使用服务质量级别 QoS2 时会采用比 QoS1 更复杂的握手和应答序列，以确保消息不会重复。QoS=2 的情况如图 1-7c 所示。

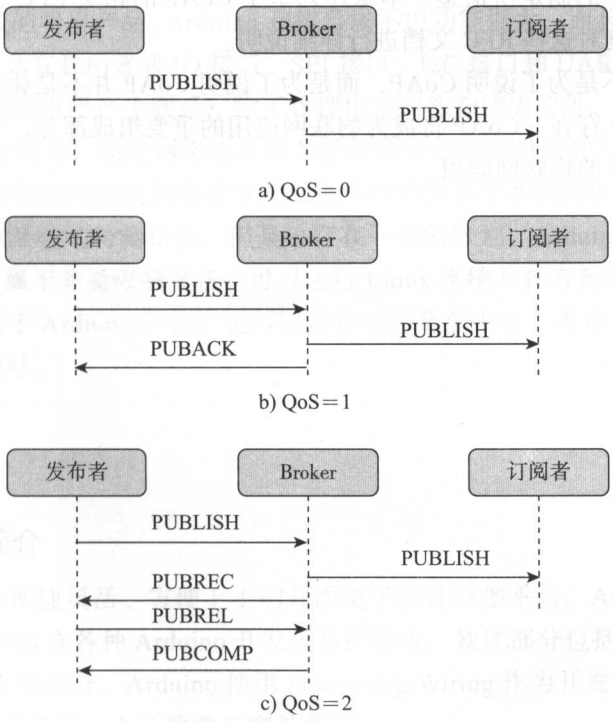


图 1-7　MQTT 消息发布服务质量

与 HTTP 和 CoAP 的请求 / 响应模式不同，MQTT 协议这样的订阅 / 发布模式总是存在三个不同的角色——发布者、代理器（Broker）和订阅者。订阅者向 MQTT 代理器订阅单个或一系列主题，发布者发布某个主题的具体消息。在 MQTT 代理器的协调下，所有订阅者将及时收到该主题的消息。MQTT 协议的工作过程经常被称为“推送”，在推送过程中某些消息必须保证稳定可靠，而某些消息允许丢失或收到重复内容。为了实现这种灵活可变的机制，MQTT 协议提供了以上三种不同的消息发布服务质量。

1.6 本章小结

本章介绍了物联网应用相关的多种协议，这些协议包括 IPv4、IPv6、6LoWPAN、HTTP、CoAP 和 MQTT 等，灵活使用这些协议将组成各种各样的物联网应用。IPv4、IPv6 和 6LoWPAN 是物联网应用的基础，IPv4 在互联网领域取得卓越的成就，但是 IPv4 地址空间短缺依然是物联网大规模应用的“痛处”，IPv6 无疑将会在物联网中取得越来越多的应用。现阶段物联网设备依然属于受限制低功耗设备，这些设备无法像非受限制设备那样完整地使用 IPv6，需要通过 6LoWPAN 头压缩技术降低协议本身的传输开销。除了这些基础协议之外，本章还讨论了 3 种应用层协议——HTTP、CoAP 和 MQTT，HTTP 和 CoAP 均采用请求 / 响应模式，而 MQTT 采用订阅 / 发布模式，其中 HTTP 已经取得广泛的应用，它的成熟经验也被 CoAP 的制定所借鉴。本章还列举了 CoAP 的相关 RFC 文档，本书的后续章节将结合具体的示例对这些 RFC 文档进行详细说明。

本章的重点并不是为了说明 CoAP，而是为了说明 CoAP 并不是凭空捏造的，也不能脱离其他网络技术独立存在。CoAP 将成为物联网应用的重要组成部分，它将与其他应用层协议一起组成多姿多彩的物联网应用。

第 2 章 Chapter 2

物联网与开源硬件

2.1 本章主要内容

本章将介绍两种在物联网领域中常用的开源硬件——Arduino 和树莓派。Arduino 和树莓派是开源硬件领域的明星产品，Arduino 和树莓派不但功能强大，而且价格便宜易于购买。Arduino 和树莓派均具有常用普通 IO 接口、SPI 接口、I2C 接口和 UART 接口等，可通过各种各样的传感器与物理世界产生联系；除了物理世界的感知能力之外，Arduino 和树莓派还具有一定的网络连接能力，通过网络连接能力把物理世界的信息传递至互联网中。Arduino 和树莓派既可以作为物联网系统的终端节点，也可以作为整个系统的中转设备或枢纽设备。

Arduino 和树莓派虽然功能相似，但是也存在一定的区别。Arduino 属于低功耗受限制设备范畴；而树莓派属于非受限制设备，可以运行 Linux 系统并具有普通 PC 的大多数功能。CoAP 不但可以运行于 Arduino 平台，也可以运行于树莓派平台。本章重点介绍几款常用的 Arduino 和树莓派型号。

2.2 Arduino

2.2.1 Arduino 简介

Arduino 是一款便捷灵活、方便上手的开源电子设计原型平台。Arduino 包含硬件和软件两部分，硬件部分包含各种 Arduino 开发板和扩展板，软件部分包括 Arduino IDE、驱动扩展库和应用扩展库等部分。Arduino 使用 Processing/Wiring 作为开发语言，该开发语言具有很多 Java 和 C 语言特性，上手简单且容易学习。

Arduino 具有以下特点：

1）跨平台：Arduino IDE 可以在 Windows、Mac OS、Linux 三大主流操作系统上运行。

2）简单清晰：Arduino IDE 基于 Processing IDE 开发。对于初学者来说，极易掌握，同时具有足够的灵活性。Arduino 语言基于 Wiring 语言开发，是对 avr-gcc 库的二次封装，不需要太多的单片机基础、编程基础，简单学习后就可以进行快速开发。

3）开放性：Arduino 的硬件原理图、电路图、IDE 软件及核心库文件都是开源的，在开源协议范围内里可以任意修改原始设计及相应代码。

4）发展迅速：Arduino 不仅仅是全球最流行的开源硬件，也是一个优秀的硬件开发平台，是硬件开发的趋势。Arduino 简单的开发方式使得开发者更关注创意与实现，更快地完成项目开发，大大节约了学习成本，缩短了开发周期。

2.2.2 常用 Arduino 型号

下面介绍两款常用的 Arduino 型号：Arduino UNO 和 Arduino 101。

1. Arduino UNO

Arduino UNO[⊖]是 Arduino USB 接口系列的最新版本，作为 Arduino 平台的参考标准模板。Arduino UNO 的核心处理器为 ATmega328，它同时具有 14 路数字输入输出口（其中 5 路可作为 PWM 输出）、6 路模拟输入、一个 16MHz 晶体振荡器、一个 USB 口、一个电源插座、一个 ICSP 插座和一个复位按钮。Arduino UNO 的外观如图 2-1 所示。

UNO 已经发布到第三版，与前两版相比该版本有以下新的特点：

1）在 AREF 处增加了两个管脚 SDA 和 SCL，支持 I2C 接口。

2）增加 IOREF 和一个预留管脚，扩展板将能兼容 5V 和 3.3V 核心板。

3）改进了复位电路设计。

4）USB 接口芯片由 ATmega16U2 替代了 ATmega8U2。

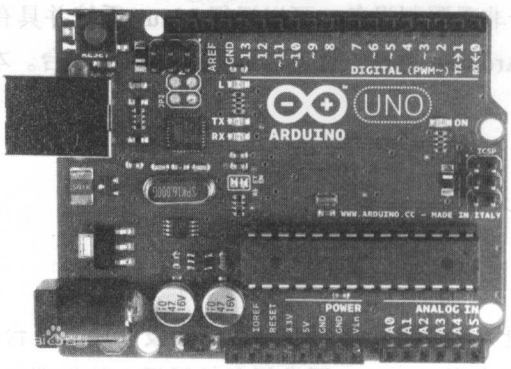


图 2-1 Arduino UNO 外观

⊖ https://www.arduino.cc/en/Main/ArduinoBoardUno。

2. Arduino 101

Arduino 101[⊖]是一个性能出色的低功耗开发板，它基于 Intel Curie 模组，价格便宜，使用简单。101 不仅有着与 UNO 一样的特性和外设，还额外增加了 Bluetooth LE 和 6 轴加速计、陀螺仪。Intel Curie 模组包含一个 x86 的夸克核心和一个 32 位的 ARC 架构核心 Zephyr，时钟频率均为 32MHz。Arduino 101 具有 14 路 I/O 口（其中 4 路可用作 PWM 输出）、6 个模拟输入、一个用于串口通信和上传程序的 USB 接口、1 个电源插座、1 个带 SPI 和 I2C 脚的 ICSP 接口。Arduino 101 的外观如图 2-2 所示。

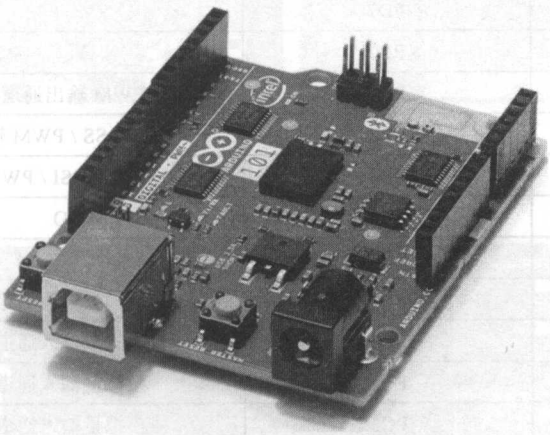


图 2-2　Arduino 101 外观

2.2.3　Arduino 扩展接口

Arduino 的成功得益于它的标准化扩展接口，通过扩展接口，Arduino 可以扩展各种各样的传感器检测功能、执行设备控制功能和网络连接功能。Arduino 可通过具有输入和输出功能的普通 IO 口控制 LED 或蜂鸣器等低功率直流设备，也可以通过操作继电器的方式间接控制大功率交流设备；Arduino 具有多路模拟量输入接口，通过这些模拟量输入接口 Arduino 可以连接多种传感器，如具有模拟量输入功能的角度传感器、三轴角速度传感器等；Arduino 具有多路 PWM 输出接口，通过 PWM 输出接口 Arduino 可以实现直流电机调速等功能；Arduino 还具有 I2C 接口和 SPI 接口，Arduino 可通过这两种常用的接口扩展各种各样的芯片，如 Arduino 可通过 SPI 接口扩展网卡芯片 W5100，通过 W5100 使 Arduino 具有有线网络功能。表 2-1 为 Arduino 扩展接口说明。

表 2-1　Arduino 扩展接口说明

编　号	单片机引脚	复用功能
0	PD0	UART RXD
1	PD1	UART TXD

⊖ https://www.arduino.cc/en/Main/ArduinoBoard101。

（续）

编　号	单片机引脚	复用功能
2	PD2	外部中断 INT0
3	PD3	外部中断 INT1 / PWM 输出通道 0
4	PD4	
5	PD5	PWM 输出通道 1
6	PD6	PWM 输出通道 2
7	PD7	
8	PB0	
9	PB1	PWM 输出通道 3
10	PB2	SPI SS / PWM 输出通道 4
11	PB3	SPI MOSI / PWM 输出通道 5
12	PB4	SPI MISO
13	PB5	SPI SCK
A0	PC0	模拟量输入通道 0
A1	PC1	模拟量输入通道 1
A2	PC2	模拟量输入通道 2
A3	PC3	模拟量输入通道 3
A4	PC4	模拟量输入通道 4 / I2C SDA
A5	PC5	模拟量输入通道 5 / I2C SCL

2.3 树莓派

2.3.1 树莓派简介

树莓派是一款具有单片机功能的小型 Linux 开发板，基于 ARM 的微型电脑主机，以 SD 或 MicroSD 卡为存储硬盘，主板周围有多个 USB 接口和一个 10/100Mbit/s 以太网接口。它可连接键盘、鼠标和网线，同时拥有 HDMI 高清视频输出接口。树莓派功能强大且接口丰富，但各种部件却整合在一张仅比信用卡稍大的主板上。

2.3.2 常用树莓派型号

市面上流行多种型号的树莓派，常见的有树莓派 2 代 B 型（见图 2-3）[⊖]和树莓派 3 代 B 型（见图 2-4）[⊜]。树莓派 2 代 B 型和树莓派 3 代 B 型在外观上并没有明显的区别，功能和性能方面树莓派 2 代 B 型和树莓派 3 代 B 型也相差不大。

⊖ https://www.raspberrypi.org/products/raspberry-pi-2-model-b/。

⊜ https://www.raspberrypi.org/products/raspberry-pi-3-model-b/。

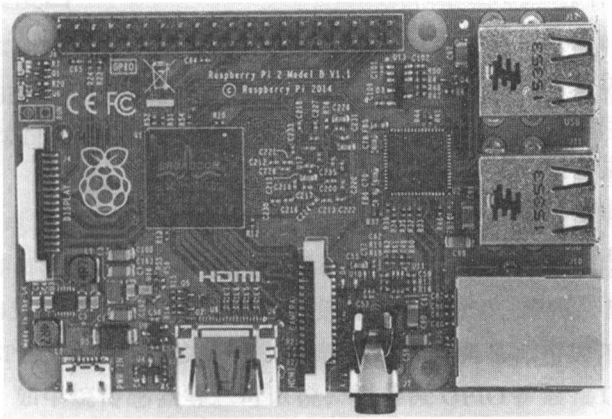


图 2-3　树莓派 2 代 B 型外观

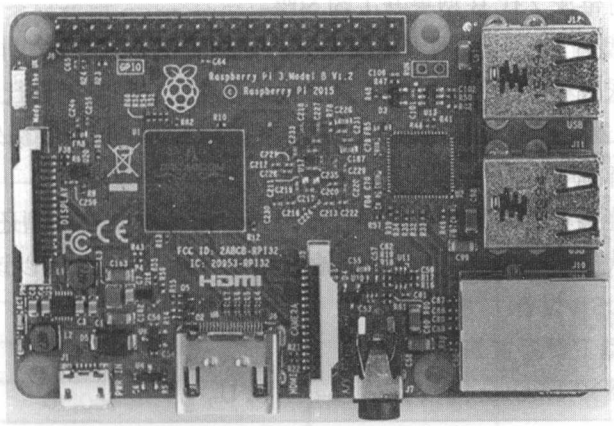


图 2-4　树莓派 3 代 B 型外观

树莓派 3 代 B 型与树莓派 2 代 B 型具有相同的 40 引脚外部扩展接口，树莓派 3 代采用了更高主频的 CPU，并内置了无线网卡和低功耗蓝牙功能，但售价与树莓派 2 代 B 型同为 35 美元，这使得树莓派 3 代 B 型越来越流行。

1. 树莓派 2 代 B 型

树莓派 2 代 B 型主板采用博通 BCM2836 的四核 CPU，主频可达 900 MHz。树莓派 2 代 B 型具有 1 GB 内存。相较于之前的树莓派版本，CPU 主频和板载内存均大幅度提升，所以树莓派的综合性能获得了广泛的认可。树莓派 2 代 B 型不但能运行全系列 ARM GNU/Linux 发行版，而且还支持 Snappy Ubuntu Core 及 Windows 10 等操作系统。

树莓派 2 代 B 型主要配置如下：

- 博通 BCM2836 900MHz 四核 ARM Cortex-A7 CPU。

- 1GB 板载内存。
- 10/100Mbit/s 自适应网卡接口。
- 4 个 USB 2.0 接口。
- Micro SD 卡插槽。
- microUSB 供电接口。
- 3.5 mm 音频输出接口。
- 40PIN GPIO 扩展接口。
- HDMI 接口。
- 摄像头接口。
- LCD 接口。

2. 树莓派 3 代 B 型

树莓派 3 代 B 型是最新型号的树莓派开发板，该主板采用 64 位 1.2 GHz 主频的四核 CPU，性能相对于树莓派 2 代 B 型提升了约 50%。

树莓派 3 代 B 型内置了无线网卡以及蓝牙，而树莓派 2 代 B 型仅内置了有线网卡，如果想要之前的树莓派版本支持无线网络或低功耗蓝牙功能，则需要自行添加 USB WiFi 适配器或 USB BLE 适配器，而对于树莓派 3 代 B 型来说则无需购买这些额外配件。所以树莓派 3 代 B 型是一款开箱即用的物联网开发工具。

除了增加网络连接功能、提升处理器性能之外，树莓派 3 代 B 型还升级了 USB 电源管理部分，树莓派 3 代 B 型 USB 部分的最大输出电流可达 2.5 A，所以该系列树莓派可支持更多更强大的外部 USB 设备。

树莓派 3 代 B 型主要配置如下：

- 博通 BCM2837 1.2GHz 四核 ARM 64 位 CPU。
- 1GB 内存。
- 板载无线网卡和低功耗蓝牙。
- 10/100Mbit/s 自适应网卡接口。
- 4 个 USB 2.0 接口。
- 升级 USB 电源，最大输出可达 2.5 A。
- Micro SD 卡插槽。
- microUSB 供电接口。
- 3.5 mm 音频输出接口。
- 40PIN GPIO 扩展接口。
- HDMI 接口。
- 摄像头接口。
- LCD 接口。

2.3.3　树莓派扩展接口

树莓派的成功不但得益于它卓越的性能、便宜的价格，还与它与生俱来的 GPIO 扩展接口有关。树莓派 GPIO 扩展接口共有 40 个引脚，这些引脚包括具有输入输出功能的普通 IO、SPI 接口、I2C 接口和 UART 接口等；这些扩展接口使树莓派能够控制各种各样的硬件。通过 GPIO 扩展接口，树莓派可非常方便地控制 LED 和蜂鸣器等低功率设备；树莓派也可以通过 GPIO 控制继电器的方式，控制直流大功率设备或者高压交流设备；树莓派 GPIO 扩展接口还具有 I2C 接口和 SPI 接口，通过这两种接口可以连接多种数字传感器，如三轴加速度传感器 ADXL345、三轴加速度 / 角速度传感器 MPU6050、温度传感器 LM75 等；树莓派通过 I2C 接口和 SPI 接口，还可以扩充 A/D 转换芯片，树莓派可通过 A/D 转换芯片连接各种模拟量传感器，如甲烷传感器等。可以说，树莓派 GPIO 扩展接口是树莓派与物理世界的一个桥梁。

1. 接口定义

表 2-2 详细说明了树莓派 GPIO 扩展接口的各引脚功能。

表 2-2　GPIO 扩展接口说明

引脚编号	功能说明	引脚编号	功能说明
1	3.3 V	21	GPIO09 / SPI MISO
2	5 V	22	GPIO25
3	GPIO02 / I2C SDA	23	GPIO11 / SPI CLK
4	5 V	24	GPIO08 / SPI CE0
5	GPIO03 / I2C SCL	25	GND
6	GND	26	GPIO07 / SPI CE1
7	GPIO04	27	ID_SD
8	GPIO14 / TXD	28	ID_SC
9	GND	29	GPIO05
10	GPIO15 / RXD	30	GND
11	GPIO17	31	GPIO06
12	GPIO18	32	GPIO12
13	GPIO27	33	GPIO13
14	GND	34	GND
15	GPIO22	35	GPIO19
16	GPIO23	36	GPIO16
17	3.3V	37	GPIO26
18	GPIO24	38	GPIO20
19	GPIO10 / SPI MOSI	39	GND
20	GND	40	GPIO21

2. 编号方式

为了更好地控制树莓派扩展接口，树莓派爱好者开发了各种计算机语言的扩展库，如使用 Python 语言开发的 RPi.GPIO 扩展库，使用 C 语言开发的 WiringPi 扩展库等。这些扩展库屏蔽了 Linux 相关驱动的实现细节，用户并不需要熟悉 Linux 驱动相关的知识也可以非常方便地控制树莓派 GPIO。这些扩展库功能相似，但是却使用了不同的编号策略。一般情况下，树莓派的引脚编号方式包括插座引脚编号方式、BCM28XX 编号方式和 WiringPi 编号方式：

- 插座引脚编号方式：编号方式侧重扩展插座引脚编号，从上到下、从左到右依次排列。这种编号方式最为简单直接。
- BCM28XX 编号方式：编号方式侧重 BCM28XX 寄存器，这种编号方式根据 BCM2835 的 GPIO 寄存器进行编号。在表 2-2 中，引脚编号为 11 的端口对应 BCM28XX GPIO17 寄存器，若采用 BCM28XX 编号方式，该端口的编号将使用 17。
- WiringPi 编号方式：编号方式侧重逻辑实现，这种编号方式把仅具有普通 IO 功能的端口从 0 开始重新编号，这种编号方式更利于代码编写。表 2-3 可以很好地反映这三种不同编号方式的区别与联系。

表 2-3　三种不同编号方式对应关系

WiringPi 编号	引脚编号	BCM28XX 编号
GPIO0	11	GPIO17
GPIO1	12	GPIO18
GPIO2	13	GPIO27
GPIO3	15	GPIO22
GPIO4	16	GPIO23
GPIO5	18	GPIO24
GPIO6	22	GPIO25
GPIO7	7	GPIO4

2.4　本章小结

本章介绍了物联网领域中常用的两种开源硬件——Arduino 和树莓派。在 2.2 节重点介绍了 Arduino UNO 和 Arduino 101 两款型号，Arduino UNO 是 Arduino 领域的入门级开发板，其采用 Atmega 系列 MCU；而 Arduino 101 是 Arduino 领域的最新型号，采用 Intel 芯片组并集成了低功耗蓝牙和 6 轴传感器功能。在 2.3 节重点介绍了树莓派 2 代 B 型和 3 代 B 型两种型号，树莓派 3 代 B 型较 2 代 B 型在整体性能上有所提升，并集成了无线 WiFi 网络和低功耗蓝牙功能。

无论是 Arduino 还是树莓派，均需要通过外部扩展接口与真实的物理世界产生联系，Arduino 和树莓派均具有标准的外部扩展接口，也就是说不同型号之间虽然性能存在差异但硬件连接方式却完全一致。这种标准化方式使得市面上出现了大量的 Arduino 或树莓派扩展板，这些扩展板集成了各种各样的传感器和执行设备，这些外部配套设备也促进了 Arduino 和树莓派的流行。

Chapter 3 第 3 章

网络技术回顾

3.1 本章主要内容

本章我们将回顾与 CoAP 相关的网络技术，从 CoAP 的历史发展轨迹可以发现，CoAP 并不是凭空产生的，也并不是完全独立设计，而是充分吸收其他协议的使用经验，针对受限制低功耗设备的特点“量身定制”的应用层协议。本章将介绍 CoAP 的“兄弟姐妹”——IPv4、IPv6、UDP、TCP 和 HTTP。现代网络一般采用分层结构设计，若参考 OSI 标准模型从上至下依次为应用层、表示层、会话层、传输层、网络层、数据链路层和物理层。通常来说，IPv4 和 IPv6 属于网络层协议，UDP 和 TCP 属于传输层协议，CoAP 和 HTTP 属于应用层协议。图 3-1 可以简单直接地展现 CoAP 和其他协议间的关系。

学习 CoAP 需要了解较多的 Web 应用方面的基础知识。如果读者还不熟悉如何使用 TCP 或 UDP 的话，建议动手尝试本章的相关示例。如果读者已经熟练掌握 Web 前端开发或者后端开发的话，那么 CoAP 学习曲线将变得相对平坦，所以这些读者可以完全忽略本章内容；如果读者对 HTTP GET 方法或 POST 方法还没有任何概念的话，那么学习 CoAP 的曲线将变得相对陡峭，所以强烈建议这些读者动手尝试本章示例。

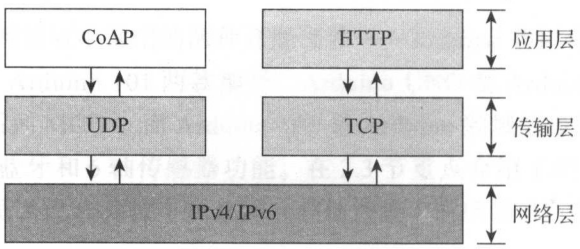


图 3-1 CoAP 的“兄弟姐妹”

本章将通过动手尝试的方式帮助读者复习 IP、TCP、UDP 和 HTTP 相关网络知识。在 3.2 节中，将通过 ping 命令访问不同的服务器，通过这种简单的方法回顾 IPv4 地址、IPv6 地址、IPv4 首部和 IPv6 首部等基础知识；在 3.3 节，将通过一个 UDP Echo 示例回顾 UDP 首部和 UDP 端口号等概念；在 3.4 节依然通过一个 TCP Echo 示例回顾 TCP 首部和 TCP 工作流程等内容；在 3.5 节，将通过一个示例网页回顾 HTTP 方法、HTTP 首部、HTTP 响应和 HTTP 媒体类型等内容。

除了 3.2 节，其他所有小节均有服务器和客户端两个角色，在相关章节的示例中，服务器的程序均运行于树莓派中，而客户端的程序均运行于 Windows 主机中。为了更好地分析 IP、UDP、TCP 和 HTTP，需要在 Windows 主机中安装较新版本的 Wireshark，通过抓取树莓派和 Windows 主机之间的网络分组数据分析相关协议的实现细节。

3.2　IP

IP 是一种在源地址和目标地址之间传输数据分包的协议，IP 是现代网络技术的重要组成部分，是整个协议族的基础，它为传输层协议 UDP 和 TCP 提供服务。IP 分为 IPv4 版本和 IPv6 版本，IPv4 为人熟知并取得了海量的应用。但是随着 IPv4 地址的枯竭，IPv6 技术进入工程师的视野。相比于 IPv4 技术，IPv6 技术有更大的地址空间、即插即用、更好的安全性和移动性等特点。与大多数工程师的一般印象相反，IPv6 协议比 IPv4 协议更加简单，借助 6LoWPAN 这样的 IPv6 头压缩技术，IPv6 可以很好地在受限制低功耗设备中使用，所以在物联网领域 IPv6 将有很好的前景。在本书第 10 章将展现一个受限制低功耗设备通过 IPv6/6LoWPAN 技术连接网络的实例。

由于 IPv6 技术并没有普及，从长远来看 IPv4 和 IPv6 将会长期并存，那么有必要同时掌握这两种技术。本节为了避免教科书式的讲解，将通过两个简单的示例说明 IPv4 和 IPv6，并说明它们之间的区别。我们将使用 ping 命令连接两个不同的服务器——腾讯服务器（www.qq.com）和百度服务器（www.baidu.com），通过 Wireshark 抓取网络数据分包分析 IPv4 和 IPv6 的相关细节，最后比较 IPv4 和 IPv6 之间的区别与联系。

3.2.1　动手尝试

本节将使用 ping 命令访问 www.baidu.com 和 www.qq.com，ping 命令一般用于测试目标主机是否可达，在 Windows 主机中使用 ping 指令，Windows 主机将会发送 ICMP 请求，若服务器收到 ICMP 请求将会立即返回 ICMP 响应，Windows 主机收到了 ICMP 响应则说明目标服务器可达。在本节相关测试环境中，www.baidu.com 仅提供了 IPv4 访问，而 www.qq.com 提供了 IPv4 和 IPv6 访问能力。

1. 测试环境说明

在本节示例中，测试 Windows 主机已经具备了 IPv4 和 IPv6 访问能力，通过 ipconfig

命令可以查看本机 IPv4 地址和 IPv6 地址，此时测试主机 IPv4 地址为 100.84.243.XXX，而 IPv6 地址为 240e:ec:4252:5095:2073:3e82:3e3:ABCD，如图 3-2 所示。

图 3-2　测试环境说明

注意　此处测试主机的 IPv4 地址隐去最后一字节，采用 XXX 代替；而测试主机 IPv6 地址隐去最后两字节，采用 ABCD 替代。

2. 尝试 IPv4 连接

启动 Wireshark，选择合适的网卡以捕获网络数据。在过滤器中输入“icmp”表示仅显示 ICMP 请求和响应数据分包。在 Windows 主机中新建控制台并在控制台中输入：

```
ping www.baidu.com
正在 Ping www.a.shifen.com [180.97.33.108] 具有 32 字节的数据:
来自 180.97.33.108 的回复: 字节=32 时间=7ms TTL=56
来自 180.97.33.108 的回复: 字节=32 时间=7ms TTL=56
来自 180.97.33.108 的回复: 字节=32 时间=6ms TTL=56
来自 180.97.33.108 的回复: 字节=32 时间=5ms TTL=56
180.97.33.108 的 Ping 统计信息:
    数据包: 已发送 = 4, 已接收 = 4, 丢失 = 0 (0% 丢失),
往返行程的估计时间(以毫秒为单位):
最短 = 5ms, 最长 = 7ms, 平均 = 6ms
```

从输出结果来看，测试 Windows 主机成功连接了百度服务器，此时百度服务器的 IPv4 地址为 180.97.33.108。在 Wireshark 中可以选中 ICMP 请求或响应的任何一个数据分包。如图 3-3 所示，此时 IP 协议的版本编号为“ Version:4”，源地址为“100.84.243.XXX”，该 IPv4 地址为 Windows 测试主机的地址，目标地址为“180.97.33.108”，该 IPv4 地址指向百度某台服务器。

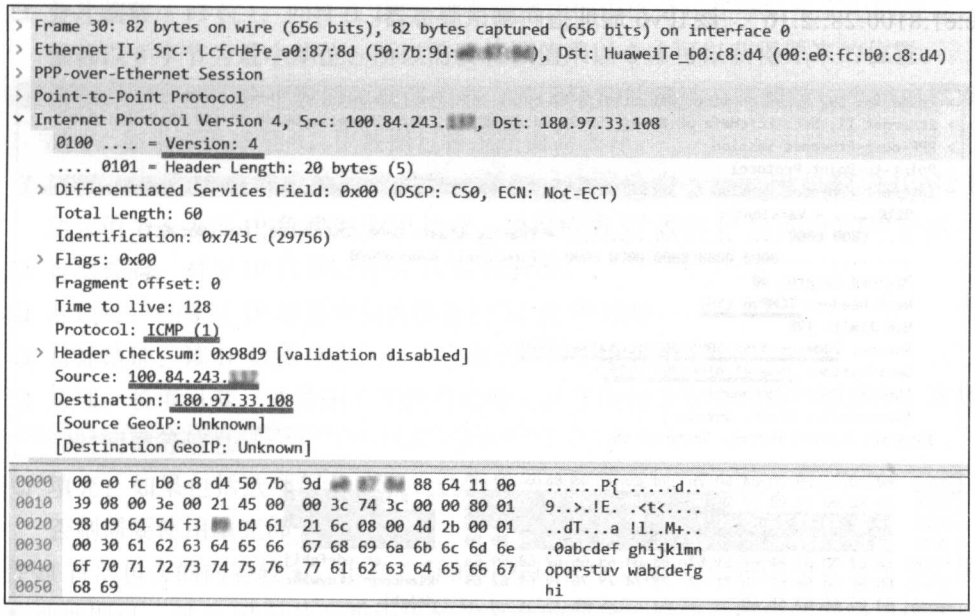


图 3-3　连接百度服务器

3. 尝试 IPv6 连接

与百度服务器不同，腾讯服务器可提供 IPv4 和 IPv6 两种访问能力。在 IPv4 和 IPv6 两种访问能力共存的情况下优先使用 IPv6 而不是 IPv4。先启动 Wireshark，选择合适的网卡以捕获网络分组数据，在显示过滤器中输入“icmpv6”表示仅显示 ICMPv6 请求和响应数据包。ICMPv6 在 IPv6 协议中非常重要，它甚至取代了 IPv4 协议中的 ARP 部分。在 Windows 主机中新建控制台并在控制台中输入：

```
ping www.qq.com
正在 Ping www.qq.com [240e:e1:8100:28::2:16] 具有 32 字节的数据:
来自 240e:e1:8100:28::2:16 的回复: 时间=5ms
来自 240e:e1:8100:28::2:16 的回复: 时间=6ms
来自 240e:e1:8100:28::2:16 的回复: 时间=5ms
来自 240e:e1:8100:28::2:16 的回复: 时间=6ms

240e:e1:8100:28::2:16 的 Ping 统计信息:
    数据包: 已发送 = 4, 已接收 = 4, 丢失 = 0 (0% 丢失),
往返行程的估计时间(以毫秒为单位):
最短 = 5ms, 最长 = 6ms, 平均 = 5ms
```

从输出结果来看，测试 Windows 主机成功连接了腾讯服务器，此时腾讯服务器的 IPv6 地址为 240e:e1:8100:28::2:16。在 Wireshark 中可选中 ICMPv6 请求或响应的任何一个数据分包。如图 3-4 所示，此时 IP 的版本编号为“Version:6”，源地址为“240e:ec:4252:5095:2073:3e82:3e3:ABCD”，该 IPv6 地址为 Windows 测试主机的地址，目标地址为

“240e:e1:8100:28::2:16”，该 IPv6 地址指向腾讯服务器。

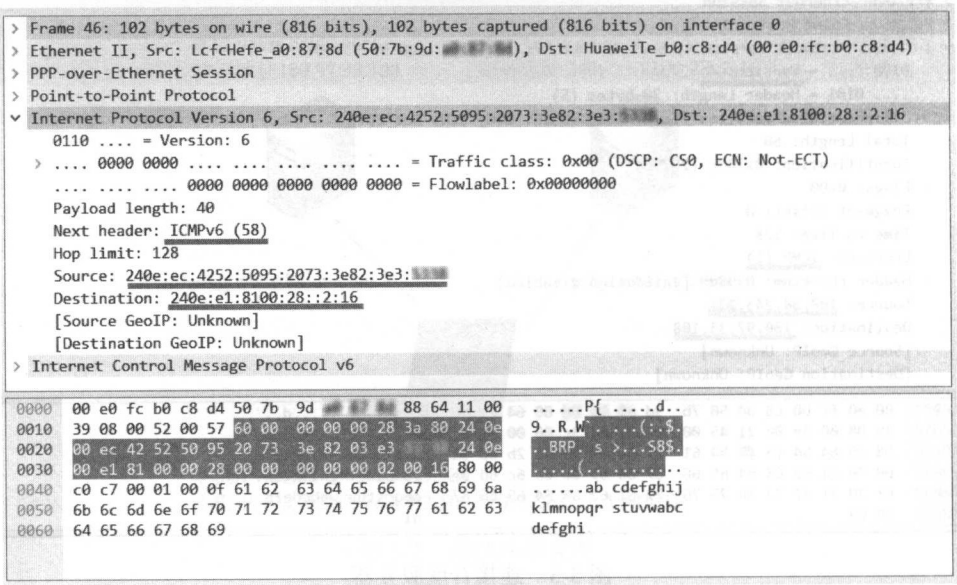


图 3-4 连接腾讯服务器

通过示例可以说明，在当前网络中 IPv4 和 IPv6 正在被同时使用，无论是 IPv4 还是 IPv6 都在发挥着自身的作用。但是随着物联网设备的普及，IPv6 将会发挥越来越多的作用。如果不熟悉 IPv6 也没有关系，市面上已经有不少关于 IPv6 的图书资料，通过阅读资料并通过 Wireshark 做些网络分析实验，也可以很快掌握 IPv6 的相关知识。经过简单的动手尝试之后我们将回顾 IPv4 和 IPv6 的基础知识。

3.2.2 IPv4 首部

IPv4 首部如图 3-5 所示，IPv4 各字段的含义说明如下：

- 版本（4 位）：该字段表示 IP 协议的具体版本，对于 IPv4 来说该值固定为 0x04。
- 首部长度（4 位）：此处的首部长度包含了首部选项与填充部分，并且以 32 位（4 字节）为最小单位。如果没有首部选项和填充部分，那么该字段的值固定为 0x05，也就是说 IPv4 的首部长度为 20 字节。
- 服务类型：表示所希望的服务质量。该字段在 IPv4 中并不常用。
- 总长度：表示 IP 数据包的总长度。该字段占 2 字节，也就是说单个 IP 数据包的最大长度为 65536 字节。
- 标识：在 IP 分片中使用，同一个 IP 数据包的所有分片具有相同的标识编号。
- 标志（3 位）：同样在 IP 分片中使用。

- 分片偏移（13 位）：同样在 IP 分片中使用，标识该分片在原始报文中的位置，分片偏移以 8 字节为最小单位，所以需要将该值乘以 8 才可以获得原来的位置。
- 生存时间：用于防止数据包在路由器之间无限期的流动。每经过一个路由器该值递减 1。如果该值减至 0，该数据包将被路由器丢弃。
- 协议：表示 IP 负载中的协议类型。例如，协议值为 1 表示 IP 负载为 ICMP 协议，协议值为 6 表示 IP 负载为 TCP 协议，协议值为 17 表示 IP 负载为 UDP 协议。
- 首部校验：对于 IP 首部计算的 16 位校验和。
- 源地址：创建该 IP 数据分包的设备的 32 位 IP 地址。
- 目标地址：该 IP 数据分包的接收设备的 32 位 IP 地址。
- 选项：长度可变，通常用于实验与诊断。该字段包含安全级别、源路径、路径记录和时间戳等信息。
- 填充：如果包括选项部分，那么选项部分的长度必须是 32 位（4 字节）的整数倍，否则需要使用填充部分。

IPv4 首部包含的内容较多，但在实际的开发过程中只需要关注 IP 协议版本号、源地址和目标地址即可。对于 IP 分片和重组等较为复杂的部分，操作系统内的网络协议栈已经为用户进行处理，在开发过程中工程师并不需要关心。

3.2.3　IPv4 地址

在设备间进行网络通信时，使用 IP 地址识别主机和路由器。为了保证设备间的正常通信，每个设备都应有正确的 IP 地址。IPv4 地址由 32 位正整数表示。IP 地址在计算机或嵌入式系统内部总是以二进制形式保存，但是人们总是不习惯于阅读和使用二进制方式，所以 IPv4 地址把 32 位 IPv4 地址分成 8 位一组，共分成 4 组，各组之间通过“.”隔开，每组通过十进制表示。192.168.1.100 便是一个合法的 IPv4 地址。

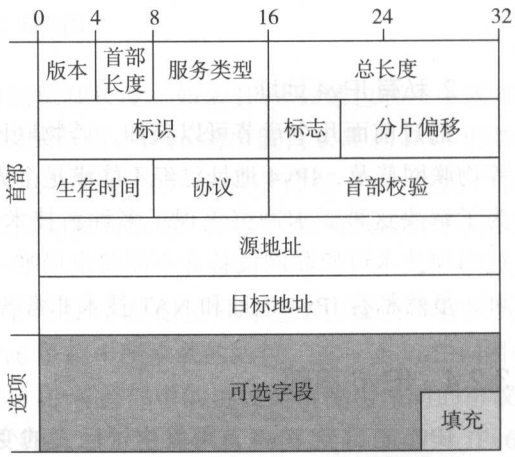


图 3-5　IPv4 首部

1. 网络 ID 和主机 ID

IPv4 地址一般由网络 ID 和主机 ID 两部分组成。网络 ID 在数据链路的每个段将配置不同的值，网络 ID 必须保证相互连接的每个段的地址不重复。而相同段内的主机必须具有相同的网络标识。这种结构非常有利于路由器的工作，路由器可以很容易地知道目标主机发送的数据在同一个网络中还是在不同的网络中，路由器可以通过比较两个 IP 地址的网络 ID 便可做出决策。

网络 ID 和主机 ID 的表示方式有两种：后缀法和子网掩码法。例如 192.168.1. 100/24，24 表示网络 ID 占该 IPv4 地址的前 24 位，此时网络 ID 为 192.168.1.0，IPv4 地址共 32 位，那么剩余的 8 位便是主机 ID，所以此时的主机 ID 为 0.0.0.100。除了后缀表示法之外，子网掩码法也非常常用，子网掩码的长度和 IPv4 地址的长度相同，若某位为比特 0 表示该位留给网络 ID，如果该位为比特 1，表示该位留给主机 ID。如果 IPv4 地址 192.168.1.100 的子网掩码为 255.255.255.0，那么它的网络 ID 为 192.168.1.0，主机 ID 为 0.0.0.100。IPv4 采用网络 ID 和主机 ID “混合排版”的方式，而 IPv6 改变了这种方式，把网络 ID 和主机 ID 完全分离。图 3-6 可以非常直观地解释 IP 地址中网络 ID 和主机 ID 的关系。

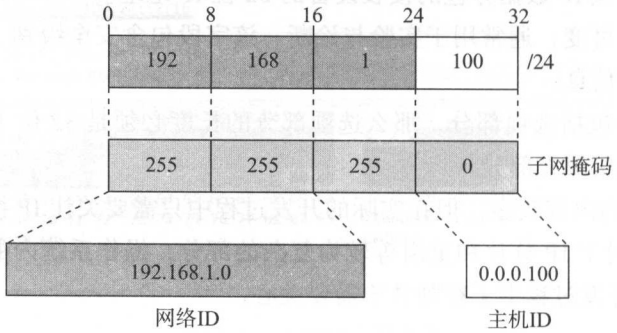


图 3-6 IPv4 地址说明

2. 私有 IPv4 地址

通过前面几个章节可以获知，若物联网设备连接网络将会消耗大量的 IP 地址，但是未等物联网普及，IPv4 地址已经不能满足个人计算机和移动手机的需求，IPv4 地址已经枯竭。为了解决这种需求冲突出现了多种新技术，其中就包括私有 IPv4 地址和 NAT 技术，通过这两项技术可使联网设备并不需要全局唯一的 IPv4 地址，仅需要在同一个域中保持唯一便可。虽然私有 IPv4 地址和 NAT 技术非常普及，但是该技术并不利于物联网设备的发展。

3.2.4 IPv6 首部

IPv6 首部较 IPv4 首部发生了巨大的变化，但是这种变化并没有使 IP 变得更复杂，反而使 IPv6 显得更简单一些。IPv6 首部如图 3-7 所示。

IPv6 首部包含以下内容：

- 版本（4 位）：表示 IP 的版本号，此处该字段为固定值 0x06。
- 流量类别：该字段与 IPv4 中的服务类型比较相似，由于服务类型在 IPv4 领域中并没有发挥实际的作用，所以原计划在 IPv6 中删除该部分，但是出于兼容和其他方面的考虑最后还是在 IPv6 中保留了该字段。
- 流标签（20 位）：流标签字段用于记录从源节点发送至多个目标节点的一系列 IPv6

数据包的序列，源节点可以利用该字段标志那些请求 IPv6 路由器进行特殊处理的数据包序列。流标签可以标识同一个流中的所有数据包，从而保证所有的数据包都可以得到 IPv6 路由器的相同处理。

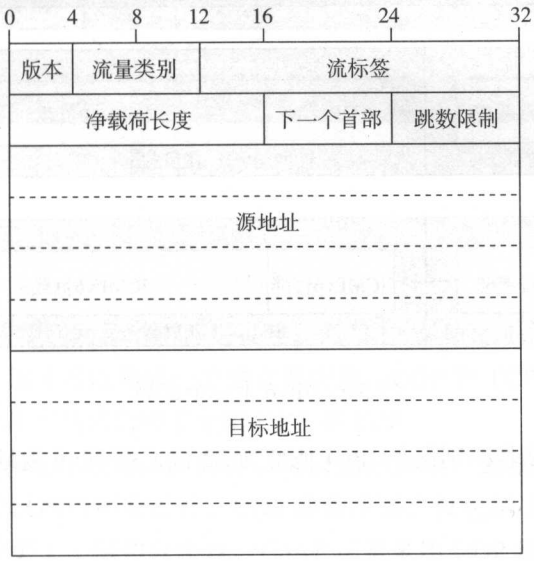


图 3-7　IPv6 首部

- 净载荷长度：该字段表示 IPv6 首部之后的负载长度。如果 IPv6 数据包有一个或多个扩展头，那么该字段也包括这些扩展头的长度。IPv6 的净载荷长度和 IPv4 的数据包总长度存在明显区别。IPv4 的数据包总长度包括 IPv4 首部和 IPv4 负载，而 IPv6 净载荷长度为 IPv6 的有效负载的长度。换句话说，IPv6 的首部长度并不需要指示，它的长度总是为 40 字节。
- 下一个首部：下一个首部有两个作用，如果 IPv6 数据包只有基本首部而没有扩展首部的话，那么下一个首部就是表示 IPv6 负载中所承载的协议，这一点与 IPv4 中的协议类型字段非常相似，而且该字段与 IPv4 首部中的协议类型使用相同的协议值，如下一个首部取值为 6 时表示 IPv6 负载为 TCP 协议，取值为 17 时表示 IPv6 负载为 UDP 协议，取值为 58 时表示 IPv6 负载为 ICMPv6 协议。图 3-8 很好地解释了下一个首部和 IPv6 负载之间的关系。
- 跳数限制：跳数限制字段与 IPv4 首部中的生存时间非常类似。该字段的值每经过一个路由器便递减 1。
- 源地址：IPv6 数据包发起设备的 128 位 IPv6 地址。
- 目标地址：该 IPv6 数据包的预计接收设备的 128 位 IPv6 地址。与 IPv4 不同，IPv6 并没有广播地址，取而代之的是 IPv6 组播地址。

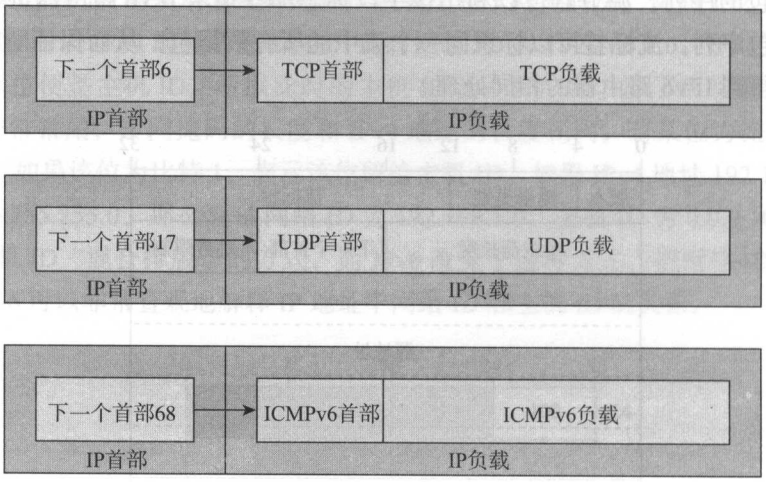


图 3-8　IPv6 下一个首部

IPv4 和 IPv6 的最显著区别便是 IPv4 地址为 32 位，而 IPv6 地址一下子增加到 128 位。

3.2.5　IPv6 地址

IPv6 地址一般采用 x:x:x:x:x:x:x:x 格式。每个 x 都是一个 16 位数，可使用 4 个十六进制数字来表示，每个 x 之间采用冒号隔开。所以 IPv6 地址包含了 8 个 16 位区域，一共为 128 位，如 2020:CA28:0000:0000:0023:0222:0000:2900。

这样的 IPv6 地址实在太长了，所以非常有必要简化 IPv6 的写法。IPv6 的简化写法可遵循两条规则——省略前导 0 和省略全 0，如图 3-9 所示。

1. 省略前导 0

所有 16 位数中的前导 0 都可以被省略，请注意该规则只适用于前导 0，而尾部的 0 绝不能被省略。那么 2020:CA28:0000:0000:0023:0222:0000:2900 将可以简化为：

2020:CA28:0:0:23:222:0:2900

2. 省略全 0

在 IPv6 地址定义中可使用双冒号 (::) 表示任意一段连续的由一个或者多个全零组成的 16 位数，采用这样的方法可以进一步简化 IPv6 地址。那么 IPv6 地址 2020:CA28:0:0:23:222:0:2900 可以进一步简化为：

2020:CA28::23:222:0:2900

无论是 IPv4 还是 IPv6 都是学习 CoAP 之前需要掌握的基础内容，由于本书篇幅有限无法详细展开，更多的内容请参考《IPv6 技术精要》和《深入解析 IPv6（第三版）》等书。

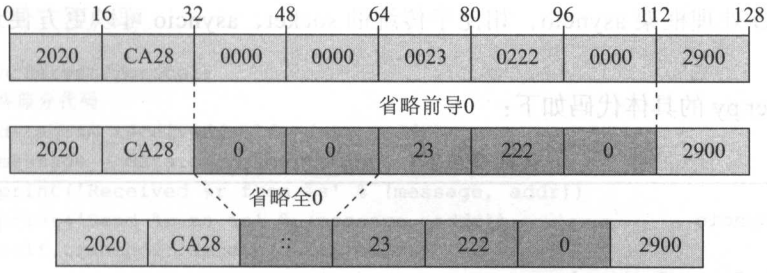


图 3-9　IPv6 地址省略写法

3.3　UDP

虽然 IP 在互联网和物联网领域中非常重要，但是 IP 本身并不能组成一个完整应用，IP 协议必须与其他协议一起才可以构成一个独立的应用。UDP 和 TCP 正是被广泛使用的传输层协议，下面我们结合几个具体的例子介绍 UDP 和 TCP。

UDP 是 User Datagram Protocal 的简称，可以翻译为用户数据协议。UDP 为那些需要简单且快速的传输层协议的应用而设计。UDP 非常简单，仅包括了端口和 IP 地址等部分，而把其他的工作都交给更上一层协议实现。CoAP 正是采用 UDP 作为传输层协议，所以学习 CoAP 之前必须要了解 UDP 的相关细节。

3.3.1　动手尝试

本节通过一个示例展现 UDP 工作的大概流程。在这个动手示例中有两个角色——UDP 客户端和 UDP 服务器，UDP 客户端可由 Windows 主机实现，而 UDP 服务器由树莓派实现，UDP 客户端和服务器端代码均使用 Python 编写。

1. 获取代码

可通过 Git 工具复制本书提供的示例代码。

```
# 新建一个名为repo的文件夹
mkdir -p repo
# clone代码仓库
git clone https://github.com/xukai871105/the_beginning_of_coap.git
# 进入目录
cd the_beginning_of_coap
# 进入网络技术回顾示例代码文件夹
cd review_demo
# 进入udp示例目录
cd udp_echo_demo
```

2. UDP 服务器实现

UDP 服务器和 UDP 客户端均使用 Python 3 编写，示例代码中使用了 Python 3.4 之后

自带的异步 IO 处理框架 asyncio，相比于传统的 socket，asyncio 可以更方便地完成网络应用开发。

udp_server.py 的具体代码如下：

代码清单3-1 udp_server.py

```
import asyncio

class EchoServerProtocol:
    def connection_made(self, transport):
        self.transport = transport

    def datagram_received(self, data, addr):
        message = data.decode()
        print('Received %r from %s' % (message, addr))
        print('Send %r to %s' % (message, addr))
        self.transport.sendto(data, addr)

loop = asyncio.get_event_loop()
print("Starting UDP server")
listen = loop.create_datagram_endpoint(
    EchoServerProtocol, local_addr=('0.0.0.0', 5683))
transport, protocol = loop.run_until_complete(listen)

try:
    loop.run_forever()
except KeyboardInterrupt:
    pass

transport.close()
loop.close()
```

udp_server.py 实现了一个简单的 UDP 服务器，服务器将把客户端发送给它的内容原样返回至客户端。

（1）创建服务器

通过 create_datagram_endpoint 方法创建一个 UDP 服务器，该方法传入两个参数：EchoServerProtocol 为一个协议实例，该实例中包含多个回调函数，用于处理建立连接、接收数据处理等事件；local_addr 用于绑定 IP 地址和端口号，'0.0.0.0' 表示侦听本机的所有网卡，5683 为侦听端口号，5683 正是 CoAP 协议的“知名”端口号。

```
listen = loop.create_datagram_endpoint(
    EchoServerProtocol, local_addr=('0.0.0.0', 5683))
```

（2）接收数据处理

在 EchoServerProtocol 中，一旦 UDP 服务器接收到网络数据，那么将会进入 datagram_received 回调函数，在该回调函数中通过 data.decode 获取客户端请求内容，最后通过 transport.

sendto 原样返回至客户端。

```
class EchoServerProtocol:
    # 省略部分代码
    def datagram_received(self, data, addr):
        message = data.decode()
        print('Received %r from %s' % (message, addr))
        print('Send %r to %s' % (message, addr))
        self.transport.sendto(data, addr)
```

3. UDP 客户端实现

UDP 客户端的代码和 UDP 服务器非常相似，读者运行该示例之前需要修改代码中的 raspberry_ip_addrss 变量，并把该变量替换为树莓派的实际 IP 地址。

udp_client.py 的具体代码如下：

代码清单3-2　udp_client.py

```
import asyncio
# replace ip address
raspberry_ip_addrss = '192.168.0.8'
class EchoClientProtocol:
    def __init__(self, message, loop):
        self.message = message
        self.loop = loop
        self.transport = None

    def connection_made(self, transport):
        self.transport = transport
        print('Send:', self.message)
        self.transport.sendto(self.message.encode())

    def datagram_received(self, data, addr):
        print("Received:", data.decode())

        print("Close the socket")
        self.transport.close()

    def error_received(self, exc):
        print('Error received:', exc)

    def connection_lost(self, exc):
        print("Socket closed, stop the event loop")
        loop = asyncio.get_event_loop()
        loop.stop()

loop = asyncio.get_event_loop()
message = "Hello World!"
connect = loop.create_datagram_endpoint(
    lambda: EchoClientProtocol(message, loop),
    remote_addr=(raspberry_ip_addrss, 5683))
```

```
transport, protocol = loop.run_until_complete(connect)
loop.run_forever()
transport.close()
loop.close()
```

udp_client.py 实现了客户端的所有功能，一旦客户端和服务器建立连接，udp_client 便把“Hello World！”字符串发送至 UDP 服务器，若 udp_client 接收到服务器返回的数据则把返回内容打印至控制台，关闭连接并最终结束程序。

（1）创建 UDP 客户端

通过 create_datagram_endpoint 方法创建 UDP 客户端，该方法传入两个参数：EchoClientProtocol(message, loop) 为一个协议示例，该实例处理网络连接、连接丢失和接收数据等事件；remote_addr=(raspberry_ip_addrss, 5683) 用于指定目标服务器地址和端口号，此处目标服务器为树莓派 IP 地址，端口号为 5683。

```
connect = loop.create_datagram_endpoint(
    lambda: EchoClientProtocol(message, loop),
remote_addr=(raspberry_ip_addrss, 5683))
```

（2）发送“Hello World！”

通过 transport.sendto(message.encode()) 把“Hello World！”发送至服务器端。

```
class EchoClientProtocol:
    # 省略若干代码
    def connection_made(self, transport):
        self.transport = transport
        print('Send:', self.message)
        self.transport.sendto(self.message.encode())
```

（3）打印接收数据

一旦接收到服务器返回的数据，那么将进入 datagram_received 回调函数，data.decode() 中便是服务器返回的“Hello World！”，打印服务器返回的数据后关闭 UDP 连接。

```
class EchoClientProtocol:
    # 省略若干代码
    def datagram_received(self, data, addr):
        print("Received:", data.decode())
        print("Close the socket")
        self.transport.close()
```

4. 执行示例代码

在树莓派控制台中输入：

```
python3 udp_server.py
```

在 Windows 控制台中输入：

```
python3 udp_client.py
```

树莓派控制台输出以下类似内容：

```
Starting UDP server
Received 'Hello World!' from ('192.168.0.3', 53495)
Send 'Hello World!' to ('192.168.0.3', 53495)
```

Windows 控制台输出以下类似内容：

```
Send: Hello World!
Received: Hello World!
Close the socket
Socket closed, stop the event loop
```

3.3.2　UDP 首部

通过运行示例代码可以发现：两个设备若需要通过 UDP 传输数据，UDP 客户端需要知晓服务器的 IP 地址和服务端口号，IP 首部中已经定义了源 IP 地址和目标 IP 地址，那么 UDP 就需要定义源端口号和目标端口号。UDP 首部如图 3-10 所示。

图 3-10　UDP 首部结构

相比于 TCP 首部，UDP 首部显得异常简单，UDP 首部仅包括源端口号、目标端口号、长度和校验和。

- 源端口号（2 字节）：表示发送进程的 16 位端口号。
- 目标端口号（2 字节）：表示接收进程的 16 位端口号。
- 长度（2 字节）：整个 UDP 数据包的长度，包含 UDP 首部和应用数据。
- 校验和（2 字节）：可选项，包括 IP 首部中的源地址与目标 IP 地址、UDP 首部。
- 应用数据：UDP 负载数据，整个 CoAP 数据分包可作为 UDP 的应用数据。

3.3.3　UDP 示例分析

1.IP 为 UPD 提供服务

现代网络采用分层设计，一般下层协议为上层协议提供服务，对于 UDP 来说，IP 是它的下层协议，IP 为 UDP 提供源 IP 地址和目标 IP 地址等信息，并把整个 UDP 数据分包作为 IP 数据分包的负载。当然 UDP 也可以为其他协议提供服务，如 CoAP。IP 与 UDP 的关系如图 3-11 所示。

2. UDP Echo 示例分析

在 UDP Echo 示例代码中，客户端和服务器相互配合采用“一问一答”的方式交换数据，这种工作模式称为“请求 / 响应”。客户端总是向服务器发起请求，若服务器接收到该

请求便根据请求的内容返回相应的响应。UDP Echo 示例正体现了这种工作模式，只是负载内的“Hello World！”字符串并没有实际含义。让我们再来分析交换“Hello World”的具体流程，该交换流程如图 3-12 所示。

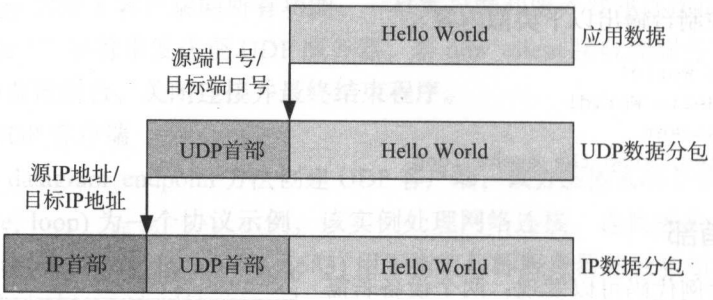


图 3-11　应用数据被封装成 UDP 数据包

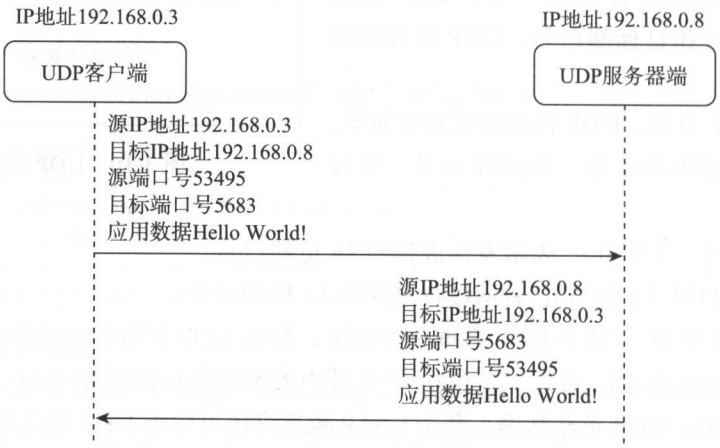


图 3-12　UDP Echo 工作流程

（1）UDP 客户端发送请求

UDP 客户端发送请求时，在 IP 首部中把本机地址作为源 IP 地址填入 IP 首部的源 IP 地址区域中，把远程主机的 IP 地址填入到 IP 首部的目标 IP 地址区域中。UDP 客户端将随机生成一个未使用的端口号，作为源端口号填入到 UDP 首部中的源端口号区域，而把远程主机的端口号填入到 UDP 首部中的目标端口号区域。

（2）UDP 服务器端返回响应

UDP 服务器填充 IP 首部和 UDP 首部的方式与 UDP 客户端非常类似，只是 IP 首部中的“源”和“目标”完全交换了位置。

3. UDP 效率分析

在 UDP Echo 示例中，UDP 有效负载为“Hello World！”共 12 字节，加上以太网首部 14 字节、IPv4 首部 20 字节，UDP 首部 8 字节，整个数据包的长度为 54 字节，请求和响应的情况几乎相同，那么此时的传输效率约为 22%。在广泛使用低功耗受限制设备的物联网领域，传感器检测结果一般都较短，可能只占用 10 字节到 20 字节，而用于数据传输的整个数据包的长度关系到系统的电量消耗。为了尽可能地提高效率，我们总是希望有效负载占整个数据分包的比例越大越好。此处 22% 的结果可能并不是一个好成绩，但是对于同样的场景应用 TCP 可能低得多。

3.4　TCP

TCP 是一种功能完备的面向连接的传输层协议，具有流控制、传输确认和重传机制。TCP 给应用提供一种可靠的、以字节流形式发送数据的方法，TCP 具有如下很多广为人知的特点：

- 面向连接：在设备之间发生数据传输之前，需要建立一个连接，数据通信双方都需要对其进行管理并在数据传送完成之后将其关闭。
- 多连接：TCP 提供了一种标识连接的方法，允许一个设备有多个连接处于打开状态而不会产生冲突。
- 全双工：一旦建立连接，数据可以在两个方向传输。

TCP 是众多应用层协议的传输工具，如 HTTP、SMTP 和 FTP 等。但 TCP 的设计也有其“历史局限性”，TCP 诞生的年代因特网并未普及，网络设备间的数据传输并不像现在这样稳定，所以 TCP 设计者通过多种技术手段保证其可靠性，这大大提高了 TCP 的复杂性。在低功耗受限制设备中，TCP 并不是唯一的选择，保证传输可靠性也可以通过应用层协议并根据实际情况量力而行。

3.4.1　动手尝试

本节通过一个示例展现 TCP 工作的大概流程，然后通过 Wireshark 获取网络数据，通过分析网络数据的方式学习 TCP 的流程与细节。该动手示例中包含两个角色——TCP 客户端和 TCP 服务器，TCP 客户端由 Windows 主机实现，而 TCP 服务器由树莓派实现。

1. 获取示例代码

可通过 Git 工具复制本书提供的示例代码。

```
# 新建一个名为repo的文件夹，若已经复制了代码，可省略新建文件夹和复制步骤
mkdir repo
# clone代码仓库
git clone https://github.com/xukai871105/the_beginning_of_coap.git
```

```
# 进入目录
cd the_beginning_of_coap
# 进入网络技术回顾示例代码文件夹
cd review_demo
# 进入tcp示例目录
cd tcp_echo_demo
```

2. TCP 服务器实现

TCP 服务器和 TCP 客户端均使用 Python 3 编写，示例代码中使用了 Python 3.4 之后自带的异步 IO 处理框架 asyncio。

tcp_server.py 的具体代码如下：

代码清单3-3 tcp_server.py

```
import asyncio

class EchoServerClientProtocol(asyncio.Protocol):
    def connection_made(self, transport):
        peername = transport.get_extra_info('peername')
        print('Connection from {}'.format(peername))
        self.transport = transport

    def data_received(self, data):
        message = data.decode()
        print('Data received: {!r}'.format(message))

        print('Send: {!r}'.format(message))
        self.transport.write(data)

        print('Close the client socket')
        self.transport.close()

loop = asyncio.get_event_loop()
coro = loop.create_server(EchoServerClientProtocol, host='0.0.0.0', port=8090)
server = loop.run_until_complete(coro)

print('Serving on {}'.format(server.sockets[0].getsockname()))
try:
    loop.run_forever()
except KeyboardInterrupt:
    pass

server.close()
loop.run_until_complete(server.wait_closed())
loop.close()
```

（1）创建 TCP 服务器

通过 create_server 方法创建一个 TCP 服务器，该方法传入三个参数：EchoServerClient-Protocol 为一个协议实例，该实例中具有多个回调函数，用于处理连接、接收数据处理等

事件；host='0.0.0.0' 用于绑定端口号；'0.0.0.0' 表示侦听本机的所有网卡，port=8090 表示 8090 为侦听端口号。

```
coro = loop.create_server(EchoServerClientProtocol, '0.0.0.0', 8090)
```

（2）接收数据处理

在 EchoServerClientProtocol 中，一旦 TCP 服务器接收到网络数据，那么将会进入 data_received 回调函数，在该回调函数中通过 data.decode 获取客户端请求内容，最后通过 transport.write 原样返回至客户端。

```
class EchoServerClientProtocol(asyncio.Protocol):
    def data_received(self, data):
        message = data.decode()
        print('Data received: {!r}'.format(message))

        print('Send: {!r}'.format(message))
        self.transport.write(data)
```

3. TCP 客户端实现

TCP 客户端的代码和服务器的代码非常相似，在运行该示例之前需要修改此处 raspberry_ip_addrss 变量，并把该变量替换为树莓派的实际 IP 地址。

tcp_client.py 的代码如下：

代码清单3-4　cp_client.py

```
import asyncio
# replace ip address here
raspberry_ip_addrss = '192.168.0.8'
class EchoClientProtocol(asyncio.Protocol):
    def __init__(self, message, loop):
        self.message = message
        self.loop = loop

    def connection_made(self, transport):
        transport.write(self.message.encode())
        print('Data sent: {!r}'.format(self.message))

    def data_received(self, data):
        print('Data received: {!r}'.format(data.decode()))

    def connection_lost(self, exc):
        print('The server closed the connection')
        print('Stop the event loop')
        self.loop.stop()

loop = asyncio.get_event_loop()
message = 'Hello World!'
coro = loop.create_connection(lambda: EchoClientProtocol(message, loop),
                              host=raspberry_ip_addrss, port=8090)
```

```
loop.run_until_complete(coro)
loop.run_forever()
loop.close()
```

（1）创建 TCP 客户端

通过 create_connection 方法创建 TCP 客户端，该方法传入两个参数：EchoClient-Protocol(message, loop) 为一个协议示例，该实例处理网络连接、连接丢失和接收数据等事件；host=raspberry_ip_addrss 用于指定远程主机 IP 地址，port=8090 用于指定远程主机端口号。

```
coro = loop.create_connection(lambda: EchoClientProtocol(message, loop),
                              host=raspberry_ip_addrss, port=8090)
```

（2）发送“Hello World”

通过 transport.write 把“Hello World”发送至服务器端。

```
class EchoClientProtocol(asyncio.Protocol):

    def connection_made(self, transport):
        transport.write(self.message.encode())
        print('Data sent: {!r}'.format(self.message))
```

（3）打印接收数据

一旦接收到服务器返回的数据，那么将进入 datagram_received 回调函数，data.decode() 中便是服务器返回的“Hello World！”，打印服务器返回的数据之后便关闭 TCP 连接。

```
class EchoClientProtocol(asyncio.Protocol):

    def data_received(self, data):
        print('Data received: {!r}'.format(data.decode()))
```

4. 执行示例代码

在树莓派控制台中输入：

```
python3 tcp_server.py
```

在 Windows 控制台中输入：

```
python3 tcp_client.py
```

树莓派控制台输出：

```
Serving on ('0.0.0.0', 8090)
Connection from ('192.168.0.3', 52908)
Data received: 'Hello World!'
Send: 'Hello World!'
Close the client socket
```

Windows 控制台输出：

```
Data sent: 'Hello World!'
Data received: 'Hello World!'
The server closed the connection
Stop the event loop
```

5. 通过 Wireshark 获取网络数据

在 Windows 主机中打开 Wireshark 选择合适的网络接口，如选择本地连接或本地无线连接，在过滤栏中输入“tcp.port==8090”用于抓取所有端口号为 8090 的 TCP 数据分包。从 Wireshark 的抓取结果可以看出，同样传输“Hello World”，TCP 相比于 UDP 需要更多的步骤。Wireshark 抓取的网络数据分包如图 3-13 所示。

tcp.port == 8090　　表达式…　+　应用此过滤器

No.	Time	Source	Destination	Protocol	Length	Info
7	2.971077	192.168.0.3	192.168.0.8	TCP	66	52908 → 8090 [SYN] Seq=0 Win=8192 Len=0 MSS=1460 WS=256 S…
8	2.971545	192.168.0.8	192.168.0.3	TCP	66	8090 → 52908 [SYN, ACK] Seq=0 Ack=1 Win=29200 Len=0 MSS=1…
9	2.971655	192.168.0.3	192.168.0.8	TCP	54	52908 → 8090 [ACK] Seq=1 Ack=1 Win=65536 Len=0
10	2.972109	192.168.0.3	192.168.0.8	TCP	66	52908 → 8090 [PSH, ACK] Seq=1 Ack=1 Win=65536 Len=12
11	2.972503	192.168.0.8	192.168.0.3	TCP	60	8090 → 52908 [ACK] Seq=1 Ack=13 Win=29248 Len=0
12	2.977103	192.168.0.8	192.168.0.3	TCP	66	8090 → 52908 [PSH, ACK] Seq=1 Ack=13 Win=29248 Len=12
13	2.978759	192.168.0.8	192.168.0.3	TCP	60	8090 → 52908 [FIN, ACK] Seq=13 Ack=13 Win=29248 Len=0
14	2.978808	192.168.0.3	192.168.0.8	TCP	54	52908 → 8090 [ACK] Seq=13 Ack=14 Win=65536 Len=0
15	2.979989	192.168.0.3	192.168.0.8	TCP	54	52908 → 8090 [FIN, ACK] Seq=13 Ack=14 Win=65536 Len=0
17	2.980337	192.168.0.8	192.168.0.3	TCP	60	8090 → 52908 [ACK] Seq=14 Ack=14 Win=29248 Len=0

图 3-13　Wireshark 获取 TCP Echo 网络数据

3.4.2　TCP 首部

前面一小节我们已经分析了 UDP 首部的具体结构，本小节我们再来了解 TCP 首部的若干细节，TCP 首部的结构相较于 UDP 要复杂得多。TCP 数据包允许一个数据分包执行多个功能，通过这种方式减少发送报文的数量，如可以在一个数据包中发送数据并确认先前接收的一个数据包。TCP 的种种特性非常适合应用，但是天下没有免费的午餐，TCP 的首部占用 20 字节，而 UDP 的首部仅有 8 字节。TCP 首部的结构如图 3-14 所示。

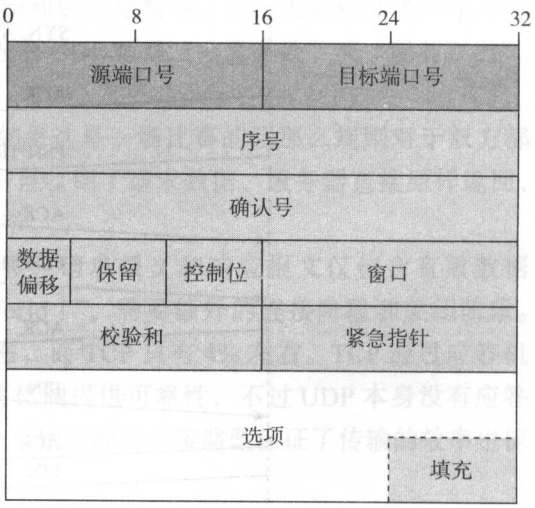


图 3-14　TCP 首部

- 源端口号（2 字节）：源设备上发送数据包进程的 16 位端口号。一般来说，对于请求数据包，它是一个客户端进程；对于应答数据包，它是一个服务器端进程。
- 目标端口号（2 字节）：目标设备上接收进程的 16 位端口号。一般来说，对于请求报文，它是一个服务器进

程；对于应答报文，它是一个客户端进程。

- 序号（4字节）：由滑动窗口确认系统使用，作为数据分包的第一字节的序号。在SYN位置为1的情况下，它表示初始序号。
- 确认号（4字节）：如果ACK位置1，该字段有效并包含设备用于对接收到的数据包进行确认的序号。
- 数据偏移（4位）：表示数据包的开始位置与TCP首部的开始处偏移多少个32位（4字节）的偏移量。该字段的值乘以4才可以得到字节形式表示的偏移量。
- 控制位（6位）：用于相关控制信息。这些控制位包括URG紧急位、ACK确认位、PUSH推送位、RST复位位、SYN同步位和FIN结束位。
- 窗口：用于流控制，表示该数据包的发送者在一定时间内能从其他设备接收的字节数。
- 校验和：用于检测错误的16位校验和。
- 紧急指针：如果URG位置1，该字段包含紧急数据后面的“正常”数据的第一字节的序列号。

3.4.3 TCP示例分析

从Wireshark的分析结果可以看出TCP的工作方式比UDP更复杂。同样传输“Hello World!”，TCP工作流程将分为建立连接、数据传输和关闭连接三个阶段。TCP的工作流程如图3-15所示。

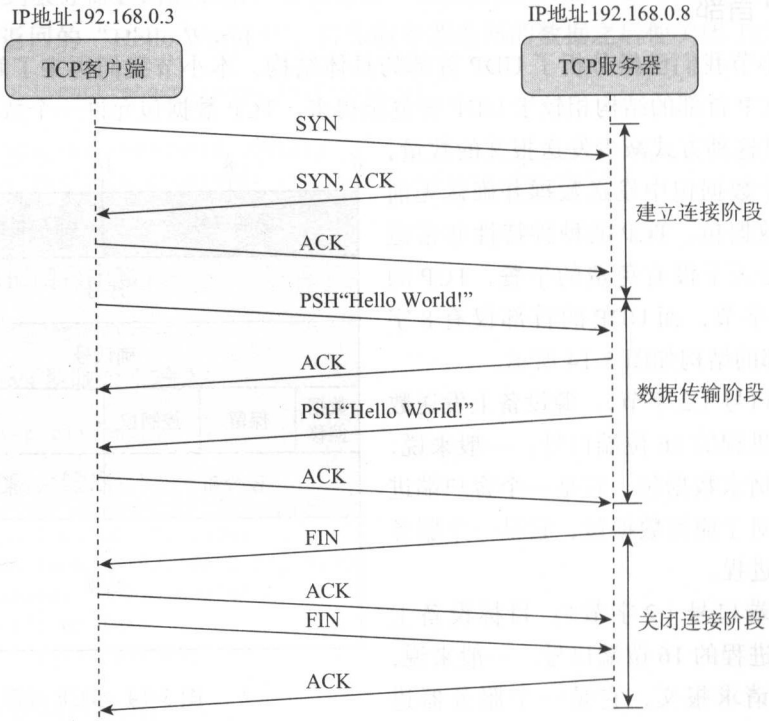


图3-15 TCP工作流程

1. 建立连接阶段

连接请求报文（SYN 报文）由一个 SYN 位置 1 的 TCP 报文构成，该报文中包含了用于连接的初始序列号。服务器收到客户端的连接请求之后必须向客户端发送一个 ACK 位置 1 的报文进行确认。由于服务器也必须将自己的初始序号通知客户端，所以为了不发送两个独立的报文，所以把报文中的 SYN 位置 1，这样就出现了一个 SYN 位和 ACK 位同时置 1 的报文。最后客户端收到服务器发来的 SYN 报文之后，同样需要对其进行确认。这就是“TCP 三次握手”。

2. 数据传输阶段

数据传输阶段与 UDP 通信类似，在本例中应用数据报文中的 PSH 位置 1，那么 TCP 将会立即把应用数据发送给服务器，服务器接收到应用数据之后将会返回 ACK 报文用于接收报文确认，表示服务器完整地接收到客户端的报文。同样，服务器原样返回客户端的请求数据也采用 PSH 发送、ACK 应答的方式。TCP 的天生应答的机制可以保证数据传输的可靠性。

3. 关闭连接阶段

关闭连接从一个 FIN 位置 1 的报文开始，一旦设备接收到 FIN 报文，必须对其进行确认，也就是说接收设备必须返回 ACK 报文。关闭连接时双方都需要发送 FIN 报文，同时双方都需要对 FIN 报文进行确认。这就是“TCP 的 4 次挥手”。

3.4.4　UDP 与 TCP 对比

在多数网络相关的技术图书中会花费大量的篇幅介绍 TCP 的各种特点以及其工作机制，如连接管理、窗口控制与重发控制、流控制和拥塞控制等。这让大多数读者产生了一个“误解”，传输层只有 TCP 才可以保证可靠性而 UDP 则是“一无是处”或“漏洞百出”。其实现实情况并不是这样的。

我们再次回到 TCP 示例和 UDP 示例中，如果这是一场比赛的话那么规则对于双方都非常公平，有效数据都是“Hello World！”，一旦收到了请求数据，服务器直接原样返回，最终有效数据为两次“Hello World！”共 24 字节。

从图 3-16 中可以发现，UDP 非常简单直接，请求报文和响应报文仅包含有效数据“Hello World！”，而 TCP 为了传输“Hello World！”，需要额外的连接阶段和关闭阶段。在这种情况下，UDP 的传输效率约为 22% 左右，而 TCP 只有 4% 左右。TCP 通过应答机制可以提高可靠性，但是 UDP 也可以通过应答机制提供可靠性，不过 UDP 本身没有应答机制需要更上一层协议来进行确认，CoAP 正是采用这种折中策略既保证了传输的效率也保证了传输的可靠性。

所以对于物联网设备来说 UDP 适用性更好，对于其他未受限制的设备来说继续使用 TCP 也无可厚非。但物联网设备光有 UDP 还不能组成一个完整的应用，还需要应用层协议

支持才行。当前已经有非常多的应用层协议采用 TCP 作为传输层协议，其中最著名的便是 HTTP。CoAP 与 HTTP 存在很多的关联性，所以学习 CoAP 之前非常有必要了解 HTTP。

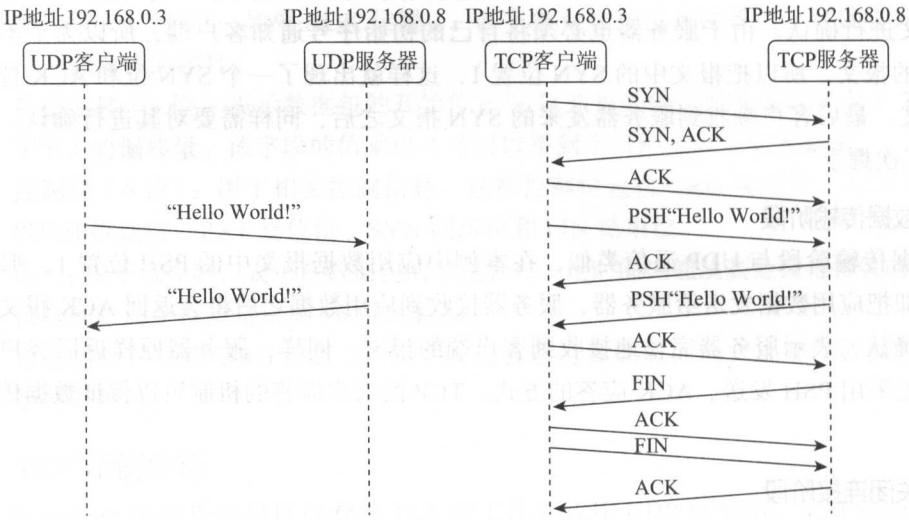


图 3-16 UDP 和 TCP 对比

3.5 HTTP

HTTP 是 Hyper Text Transfer Protocol（超文本传输协议）的缩写。它的发展是万维网协会（World Wide Web Consortium）和 IETF（Internet Engineering Task Force）合作的结果。到目前为止，已经有很多个版本的 HTTP 被成功应用，包括 HTTP 1.0 和 HTTP 1.1，HTTP 2.0 也已发布并逐渐推向应用。相比于 HTTP 1.X，HTTP 2.0 有非常多的优势，但是本节依然以回顾 HTTP 1.x 为主。

CoAP 借鉴了 HTTP 的大量成功经验，如果已经熟练掌握 HTTP 或 Web 开发的话，那么学习 CoAP 将变得相对简单。如果还没有接触过 HTTP 应用或 Web 开发的话，强烈建议动手尝试本节的例子。

3.5.1 动手尝试

在入门示例中，依然使用树莓派作为服务器，与 TCP 和 UDP 入门示例不同，此处树莓派将作为一个 HTTP 服务器。相比于 TCP 和 UDP 服务器角色，HTTP 服务器需要承担更多的责任。此时浏览器将作为客户端，在浏览器的地址栏中输入树莓派的 IP 地址加上一个约定好的端口号便可访问树莓派提供的 HTTP 服务。HTTP 服务通常被称为 Web 服务，在树莓派中建立一个 Web 服务有非常多的方法，有很多成熟的 Web 框架可以使用，如基于 Python 的 Flask 框架，或者基于 Node.js 的 Express 框架。动手尝试示例试图通过现象看本

质，所以使用哪个框架并不重要。HTTP 动手尝试示例的网络结构如图 3-17 所示。

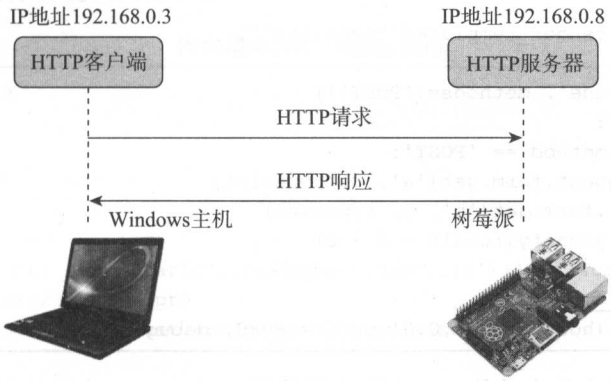


图 3-17　HTTP 动手尝试入门示例

1. 安装 Python Flask

树莓派中已经默认安装了 Python 2 和 Python 3，但并没有安装 Flask 框架，所以要给树莓派增加 Web 功能需要通过 pip 工具为 Python 3 安装 Flask 框架。在树莓派控制台中输入以下指令便可完成 Flask 框架安装。

```
sudo pip3 install flask
```

2. 获取示例代码

为了减少读者编写代码的时间，可直接复制本书示例代码。HTTP 示例代码位于 review_demo/http_demo 目录中。

```
# 新建一个名为repo的文件夹，若已经复制了代码，可省略新建文件夹和复制步骤
mkdir repo
# clone代码仓库
git clone https://github.com/xukai871105/the_beginning_of_coap.git
# 进入目录
cd the_beginning_of_coap
# 进入网络技术回顾示例代码文件夹
cd review_demo
# 进入http示例目录
cd http_demo
```

3. Web 后端实现

Web 应用总是可分为前端实现和后端实现两部分。后端一般用于处理用户请求、从数据库获取数据、进行逻辑处理等工作。Web 后端的实现代码 app.py 如下：

代码清单3-5　app.py

```
# -*- coding: utf-8 -*-
from flask import Flask, jsonify, render_template, request
app = Flask(__name__)
```

```
@app.route("/")
def index():
        return render_template('index.html')

@app.route('/value', methods=['POST'])
def cal_value():
    if request.method == 'POST':
        a = request.form.get('a', 0, type=int)
    b = request.form.get('b', 0, type=int)
        return jsonify(result = a + b)

if __name__=="__main__":
        app.run(host = '0.0.0.0',port = 8080, debug = True)
```

app.py 主要有两个路由功能：一个路由可使用 HTTP GET 方法访问，返回一个 html 页面，该路由可理解为“主页面”；另一个路由可使用 HTTP POST 方法访问，返回两个参数 a 和 b 相加的结果。

（1）渲染 index.html

若使用浏览器客户端访问根路由“/”，那么 Web 服务器将渲染一个 index.html 文件并把该文件返回给浏览器，index.html 保存在 http_demo 的 templates 文件夹中。

```
@app.route("/")
def index():
return render_template('index.html')
```

（2）计算参数 a 和 b 之和

若浏览器客户端访问路由“/value”，那么 Web 服务器将取出客户端请求中的两个参数 a 和 b，计算两者之和并把结果作为返回内容。浏览器请求中把参数 a 和参数 b 通过“a=12&b=34”的方式组织起来，而 Web 服务器使用“{“result”:46}”这样的 JSON 格式包装响应。

```
@app.route('/value', methods=['POST'])
def cal_value():
  if request.method == 'POST':
    a = request.form.get('a', 0, type=int)
    b = request.form.get('b', 0, type=int)
    return jsonify(result = a + b)
```

（3）运行 Web 服务器

指定 Web 服务的端口号为 8080，并通过 debug = True 启动 Flask 的调试功能。

```
if __name__=="__main__":
    app.run(host = '0.0.0.0',port = 8080, debug = True)
```

4. Web 前端实现

在入门示例中仅包括一个前端页面 index.html，该文件位于 templates 文件夹中。index.

html 中包含几个 HTML 元素和一小段 JavaScript 代码，而 JavaScript 代码基于 JQuery 编写。index.html 的具体实现代码如下：

代码清单3-6　index.html

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>HTTP Demo</title>
<!-- 插入jquery -->
<script src="{{url_for('static', filename='jquery.js')}}"></script>
<script type=text/javascript>
var $SCRIPT_ROOT = {{request.script_root|tojson|safe}};
</script>
<script type=text/javascript>
$(function() {
    function submit_form(e) {
        $.ajax({
            type: "post",
            url: $SCRIPT_ROOT + '/value',
            data: {
                a: $('input[name="a"]').val(),
                b: $('input[name="b"]').val(),
                now: new Date().getTime()
            },
            success: function(data) {
                $('#result').text(data.result);
            }
        });
    }
    // 绑定click事件
    $('#calculate').bind('click', submit_form);
});
</script>
</head>
<body>
<p>
    <input type=text size=5 name=a> +
    <input type=text size=5 name=b> =
    <span id=result>?</span>
</p>
<p><input type="button" id="calculate" value="计算"></p>
</body>
</html>
```

（1）页面 HTML 元素

index.html 页面中包括的 HTML 元素很少，主要有两个输入框，用于输入参数 a 和参数 b。另外还有一个按钮，点击按钮将会触发一次 Ajax 异步请求。这个简单的页面如图 3-18 所示。

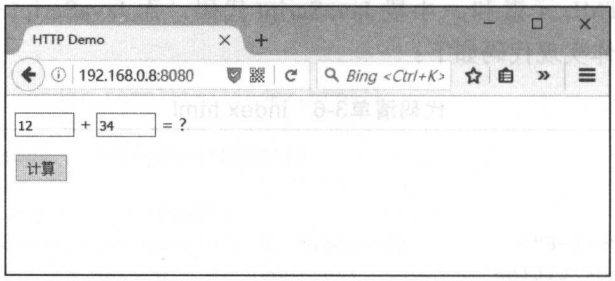


图 3-18　index.html

（2）执行异步请求

在 index.html 中包含一段 JavaScript 代码，一旦 ID 编号为 calculate 的 HTML 元素被单击，将会调用 submit_form 函数。在 submit_form 函数中取出输入框中 a 和 b 的内容，并通过 Ajax 方式提交，最后把服务器的返回结果显示到 ID 编号为 result 的 HTML 元素中。

```
$(function() {
    function submit_form(e) {
        $.ajax({
            type: "post",
            url: $SCRIPT_ROOT + '/value',
            data: {
                a: $('input[name="a"]').val(),
                b: $('input[name="b"]').val(),
                now: new Date().getTime()
            },
            success: function(data) {
                $('#result').text(data.result);
            }
        });
    }
    // 绑定click事件
    $('#calculate').bind('click', submit_form);
});
```

5. 执行示例代码

（1）树莓派中启动服务器

在树莓派控制台输入：

```
python3 app.py
```

（2）获取运算结果

在 Windows 主机中打开浏览器，在浏览器地址栏中输入 Web 服务器的 IP 和应用端口号，如 http://192.168.0.8:8080。浏览器将会获得 index.html 页面内容，在该页面中填入参数 a 和参数 b 的值，如参数 a 输入框中填入 12，参数 b 输入框中填入 34，点击计算按钮之后可获得 Web 服务器返回的计算结果 46。整个过程如图 3-19 所示。

图 3-19　获取运算结果

6. 运行分析

HTTP 相关的网络数据包分析要比 UDP 和 TCP 数据包分析更复杂一些。分析 HTTP 请求响应一般要借助浏览器的调试工具和 Wireshark，更多的时候需要灵活地同时使用两种工具。在 Firefox 浏览器和 Chrome 浏览器中可使用快捷键〈 Ctrl+Shift+I 〉进入开发者界面，在网络选项卡中可获取 HTTP 请求和响应的所有信息。如图 3-20 所示为使用 Firefox 浏览器分析 HTTP 请求响应。

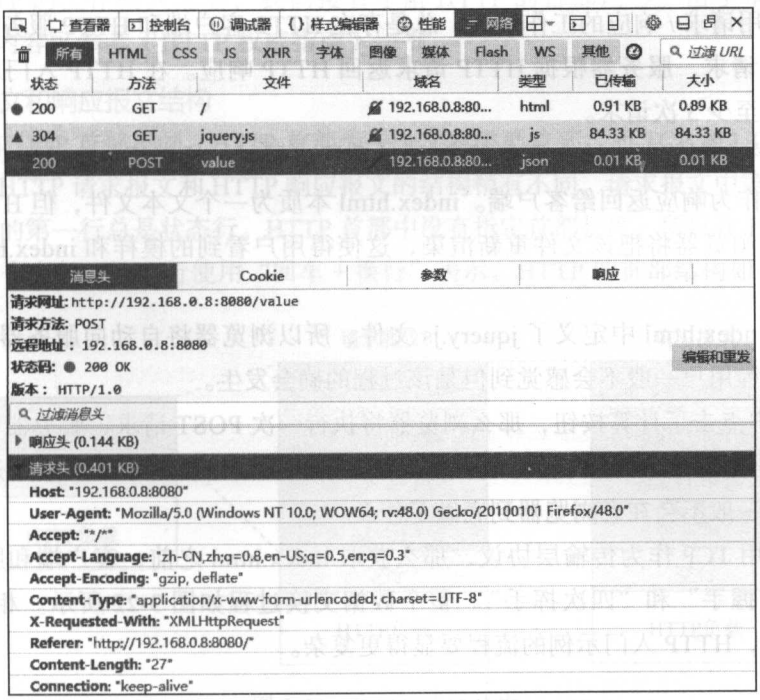


图 3-20　使用 Firefox 浏览器分析 HTTP 请求响应

3.5.2 HTTP 工作模式

Web 开发包括前端、后端和数据存储等多个部分，随着 Web 的飞速发展出现了各种各样的开发框架，这些开发框架给开发者带来了越来越多的便利，但也使得代码离 HTTP 本身越来越远。从本质上来说 HTTP 设计原则和 TCP 或者 UDP 非常相似，HTTP 也由首部和负载两部分组成，HTTP 首部决定了 HTTP 本身的相关特性，而 HTTP 负载则用于传输更上层的应用协议，这些内容包括 HTML、CSS、JavaScript、JSON 和 XML 等内容。HTTP 采用 TCP 作为传输层协议，其与 TCP、IP 的关系如图 3-21 所示。

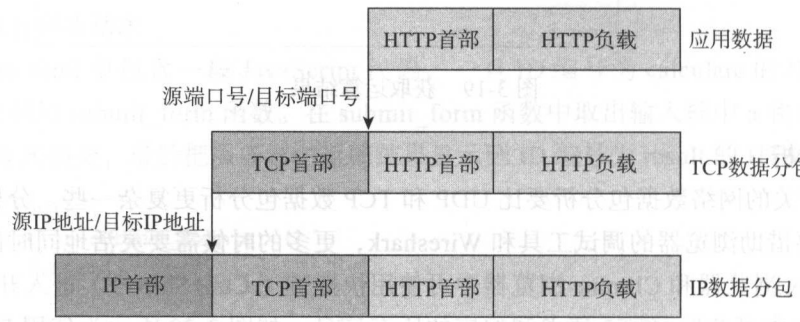


图 3-21 HTTP 与 TCP IP 的关系

HTTP 采用请求 / 响应的工作模式，总是存在 HTTP 客户端和 HTTP 服务器，客户端负责发起 HTTP 请求，服务器根据 HTTP 请求返回 HTTP 响应。在 HTTP 入门示例中，客户端总共发起了至少 3 次请求。

1）当用户在浏览器中输入服务器 IP 地址和端口号时，执行第一次 GET 请求，服务器把 index.html 作为响应返回给客户端。index.html 本质为一个文本文件，但 HTTP 中称它为超文本文件，浏览器将把该文件重新渲染，这使得用户看到的模样和 index.html 本身完全相同。

2）由于 index.html 中定义了 jquery.js 文件，所以浏览器将自动向服务器获取 jquery.js 文件。这个过程用户一般不会感觉到但是该过程的确会发生。

3）当用户点击了计算按钮，那么浏览器将执行一次 POST 请求，服务器将返回计算结果。计算结果采用 JSON 格式包装，经过处理之后在页面内显示结果。这个过程用户可以感觉到，但是一般不会在意浏览器到底做了什么。

HTTP 采用 TCP 作为传输层协议，那么获取 index.html 之前，客户端和服务器之间还发送了“三次握手”和“四次挥手”。整个数据交换过程如图 3-22 所示。相比于 UDP 和 TCP 入门示例，HTTP 入门示例的流程要显得更复杂。

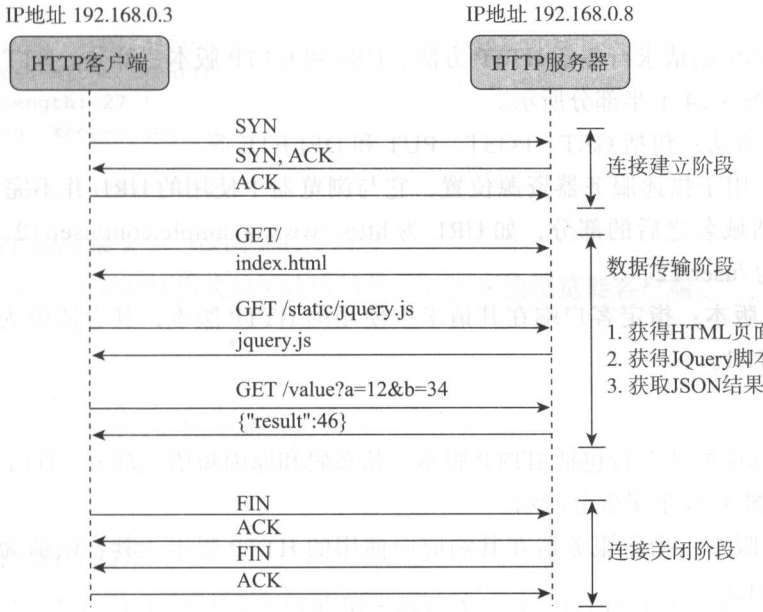


图 3-22　HTTP 客户端和服务器数据交换过程

3.5.3　HTTP 首部

了解 HTTP 的工作模式之后，我们接着了解 HTTP 的首部内容。由于本书篇幅限制，HTTP 的更多内容可参考《HTTP 权威指南》一书。

1. 请求报文和响应报文结构

与 TCP 和 UDP 首部不同，HTTP 首部为一个文本类型首部，而 TCP 和 UDP 首部为一个二进制首部。HTTP 请求报文和 HTTP 响应报文的结构稍有不同，请求报文中第一行总是请求行而响应报文的第一行总是状态行。HTTP 首部中没有指定首部长度，所以在 HTTP 首部和负载之间存在一个空行，该空行使用“回车 + 换行”表示。HTTP 的首部结构如图 3-23 所示。

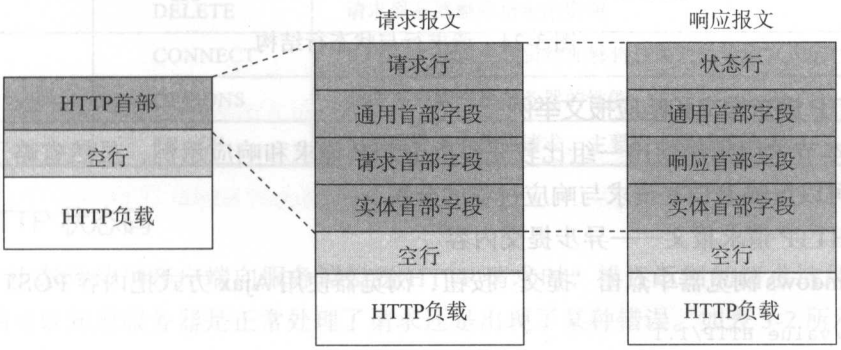


图 3-23　HTTP 首部结构

2. 请求行

HTTP 首部中的请求行包括 HTTP 方法、URI 和 HTTP 版本三部分。HTTP 首部中的请求行的结构如图 3-24 上半部分所示。

- HTTP 方法：包括 GET、POST、PUT 和 DELETE 等。
- URI：用于描述服务器资源位置，它与浏览器中使用的 URL 并不完全等同，URI 仅包括域名之后的部分，如 URL 为 http://www.example.com/user/12，那么此时的 URI 为 /user/12。
- HTTP 版本：指定客户端在其请求中使用的 HTTP 版本，其合法值为 HTTP/1.0 或 HTTP/1.1。

3. 状态行

HTTP 首部中的状态行包括 HTTP 版本、状态码和原因短语三部分。HTTP 首部中的状态行的结构如图 3-24 下半部分所示。

- HTTP 版本：指定服务器在其响应中使用的 HTTP 版本，其合法值为 HTTP/1.0 或 HTTP/1.1。
- 状态码和原因短语：状态码反映服务器处理客户端请求的结果，该信息用一个 3 位数字表示，而原因短语使用文本形式表示。

下面是一个非常受欢迎的状态行，如果获取该状态行表示服务器成功处理了请求。

```
HTTP/1.0 200 OK
```

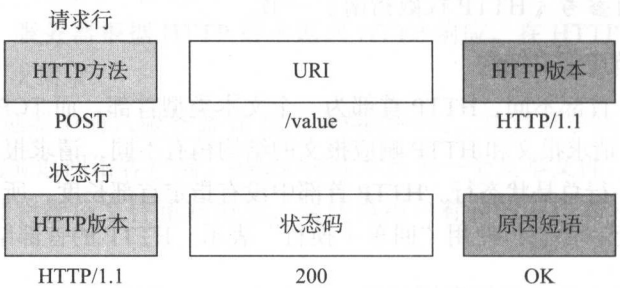


图 3-24　请求行与状态行结构

4. HTTP 请求报文和响应报文举例

结合本节入门示例给出一组比较完整的 HTTP 请求和响应示例，虽然省略了部分内容但是依然可以反映 HTTP 请求与响应过程的全貌。

（1）HTTP 请求报文——异步提交内容

在 Windows 浏览器中点击“提交”按钮，浏览器使用 Ajax 方式把内容 POST 至服务器。

```
POST /value HTTP/1.1
Host: 192.168.0.8:8080
User-Agent: Mozilla/5.0
```

```
Accept: */*
Content-Type: application/x-www-form-urlencoded; charset=UTF-8
X-Requested-With: XMLHttpRequest
Content-Length: 27
Connection: keep-alive

a=12&b=34&now=1472309742455
```

（2）HTTP 响应报文——返回 JSON 内容

Web 服务器通过 JSON 格式包装响应结果，并返回给浏览器客户端。

```
HTTP/1.0 200 OK
Content-Type: application/json
Content-Length: 14
Server: Werkzeug/0.11.10 Python/3.4.2

{"result":46}
```

3.5.4　HTTP 请求方法

根据 HTTP 标准，HTTP 请求可以使用多种请求方法。HTTP 1.0 标准定义了三种请求方法：GET、POST 和 HEAD 方法。HTTP 1.1 标准又新增了五种请求方法，即 OPTIONS、PUT、DELETE、TRACE 和 CONNECT 方法。如表 3-1 所示。

表 3-1　HTTP 请求方法

序　号	请求方法	描　述
1	GET	请求指定资源，并返回实体主体
2	HEAD	类似于 GET 请求，但返回的响应中没有具体的内容，可用于获取 HTTP 报文首部
3	POST	向指定资源提交数据（例如提交表单或者上传文件）。数据被包含在 HTTP 请求负载中。POST 请求可能导致新的资源的建立和 / 或已有资源的修改
4	PUT	客户端向服务器请求的数据更新指定的资源
5	DELETE	请求服务器删除指定的资源
6	CONNECT	HTTP1.1 协议中预留给能够将连接改为管道方式的代理服务器
7	OPTIONS	允许客户端查看服务器的性能
8	TRACE	回显服务器收到的请求，主要用于测试或诊断

3.5.5　HTTP 状态码

HTTP 状态码是当客户端向服务器发送 HTTP 请求时，描述返回的请求结果。借助状态码客户端可以知道服务器是正常处理了请求还是出现了某种错误。如表 3-2 所示为 HTTP 状态码和原因短语。

表 3-2 HTTP 状态码和原因短语

状态码类型	状态码与原因短语	说　明
信息报文	100 Continue	客户端继续其请求
	101 Switching Protocols	服务器同意带有更新首部的客户端请求，改变在该连接上正在使用的应用层协议
成功	200 OK	成功
	204 No Content	成功，但不返回任何实体的主体部分
	206 Partial Content	成功执行了一个范围请求
重定向	301 Moved Permanently	
	302 Found	永久性重定向，响应报文的 Location 首部应该有该资源的新 URL
	304 Not Modified	客户端发送附带条件的请求（请求首部中包含如 If-Modified-Since 等指定首部）时，服务端有可能返回 304，此时响应报文中不包含任何报文主体
客户端错误	400 Bad Request	请求报文中存在语法错误
	401 Unauthorized	需要认证
	404 Not Found	服务器上无法找到请求的资源
服务器错误	500 Internal Server Error	服务端在执行请求时发生了错误
	503 Service Unavailable	服务器暂时无法提供服务，可以包含 Retry-After 首部

3.5.6 HTTP 首部字段

HTTP 首部字段是构成 HTTP 报文的要素之一，在客户端与服务器之间的 HTTP 请求响应过程中，无论是请求还是响应都会使用首部字段。HTTP 首部字段使用首部字段名和字段值构成，中间使用冒号“:”风格，例如：

```
首部字段名：首部字段值
```

HTTP 首部字段按照实际用途分为通用首部字段（General Header Fields）、请求首部字段（Request Header Fields）、响应首部字段（Response Header Fields）和实体首部字段（Entity Header Fields）4 个部分：

- 通用首部字段：包括 Connection、Data 和 Upgrade 等。
- 请求首部字段：包括 Host、User-Agent、Accept、If-Match 和 Range 等。
- 响应首部字段：包括 Age、Location 和 Set-Cookie 等。
- 实体首部字段：包括 Content-Encoding、Content-Length、Content-Type、Content-Range 和 Etag 等。

3.5.7 HTTP 的优势与问题

虽然 HTTP 已经在互联网领域取得了非常广泛的应用，但是它也存在一些被人忽略的

问题。为了在物联网领域借鉴 HTTP 的优势并克服 HTTP 的缺陷，CoAP 应运而生。

1. 优势

HTTP 的确具有不少优势，而且这些优势依然可以在物联网领域继续延续。

（1）简单的工作模式

HTTP 采用请求 / 响应的工作模式，这样一问一答的方式使得系统设计更加简单。HTTP 请求总是由客户端发起，服务器对请求做出响应。HTTP 采用无状态设计方式，HTTP 请求时总是带上一些必要的信息，这使得服务器不必保留多余的关于客户端的信息。这种无状态设计方式降低了服务器的实现复杂度。

（2）完整的方法定义

HTTP 定义了多种请求方法，在 REST 风格下这些请求方法和数据库操作中的“增删改查”建立了完整的对应关系。在 REST 风格下服务器 API 设计变得越来越简单清晰，这也在软件实现层面降低了复杂度。

（3）合理的状态码设计

HTTP 中设计了 4 种不同类型的状态码，这些状态码帮助应用完成了最基本的错误处理。通过这种方式也可降低具体应用的设计复杂程度。

（4）友好的媒体类型支持

HTTP 负载类型定义非常灵活，HTTP 负载可以是各种各样的媒体类型，这些媒体类型包括文本类型、图片类型、视频类型或者任何一种二进制类型。HTTP 的这种定义方式把更多的“想象空间”留给具体应用，这使得现阶段的 Web 应用变得如此丰富多彩。

2. 问题

当然 HTTP 并不是完美的，随着时代的发展人们对 HTTP 也有了更多现实的认识。这些问题包括冗长的文本首部和单向传输方式。这些问题的暴露也导致新的协议的产生，如 CoAP 和 WebSocket。近年来 HTTP 本身也有不少的改进，HTTP 2.0 版本是一个全新的 HTTP 版本。

（1）冗长的文本首部

HTTP 采用文本首部的设计方式，这些文本首部虽然便于人类阅读，但是也使得 HTTP 首部变得越来越冗长。例如 Content-Type:application/json，该首部字段传输过程中将占用 29 字节。另外 HTTP 采用无状态方式设计，这使得每次请求的 HTTP 首部都很长，这种方式在物联网领域肯定是一种浪费。在物联网领域，设备上传传感器结果可能只有十几或二十几字节，教条地使用 HTTP 将会把宝贵的带宽浪费在传输 HTTP 首部上而不是应用数据本身。

（2）单向传输方式

HTTP 请求 / 响应模式也限制了 HTTP 的传输方向，服务器不能主动地向客户端传输数据。WebSocket 可以解决 HTTP 的这种问题，并与 HTTP 配合默契。WebSocket 建立起浏览

器和服务器之间的双向通道。另外 HTTP 2.0 也解决了该问题，允许服务器直接向客户端传输数据。

3.6 本章小结

本章回顾了互联网领域被广泛使用的网络技术，这些技术包括 IP、UDP、TCP 和 HTTP。IP 部分回顾了 IP 的两个版本 IPv4 和 IPv6，IPv4 虽然在现阶段被广泛应用，但是由于地址枯竭并不能满足物联网设备的需求，可以预见在不久的将来 IPv6 将在物联网设备中取得一定的成绩。本章还回顾了传输层的两个重要协议——UDP 和 TCP，无论是 UDP 还是 TCP 小节，都通过一个实际的例子说明它们的工作方式。UDP 设计简单，而 TCP 包括了“三次握手”和“四次挥手”，并且每个 TCP 报文都需要 ACK 报文作为响应，TCP 表现出良好的可靠性。但通过本章中的 TCP 和 UDP 比较，在传输相同内容的情况下，UDP 比 TCP 效率更高，可以说 UDP 更适合物联网设备。本章最后还回顾了 HTTP，HTTP 应用非常广泛，该小节回顾了 HTTP 请求方法、HTTP 状态码和请求首部字段等内容。虽然 HTTP 在互联网领域占有统治地位，但它的冗长首部也使得它在物联网领域力不从心。

IPv4、IPv6、UDP、TCP 和 HTTP 都是 CoAP 的“兄弟姐妹”，学习与回顾这些内容对 CoAP 的学习非常有价值，此时 CoAP 可以使用一个比较直观的公式说明它与 IPv4、IPv6、UDP 和 HTTP 的关系，如图 3-25 所示。

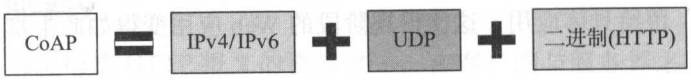
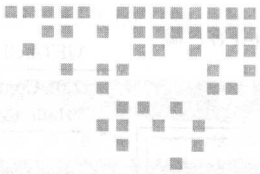


图 3-25 CoAP 与 IPv4、IPv6、UDP 和 HTTP 的关系



第 4 章 Chapter 4

CoAP 快速入门

4.1 本章主要内容

本章试图通过一个简单示例为大家开启 CoAP 旅程，这个简单的示例包括一台已经安装 Firefox（火狐）浏览器的个人电脑。入门示例将会在 Firefox 浏览器中安装一个名为 Copper 的插件。该插件提供完整的 CoAP 客户端调试功能，只需知晓 CoAP 服务器的 IP 地址或域名，借助该插件便可向服务器提交 CoAP 请求，除了提交 CoAP 请求之外还可以设置 CoAP 请求参数。入门示例使用 Arduino UNO 作为 CoAP 服务器。Arduino UNO 仅拥有 2KB 大小的内存，虽然只有 2KB 内存空间，但其资源也可以满足一个 CoAP 服务器的基本要求。为了使 Arduino UNO 具有联网功能，还需要为 Arduino UNO 增加网络扩展板，Arduino 网络扩展板有很多种型号，本章推荐使用 W5100。

CoAP 入门示例的具体组成如图 4-1 所示，安装有 Firefox 浏览器的 Windows 主机作为 CoAP 客户端，而 Arduino UNO 作为 CoAP 服务器。CoAP 服务器提供数量有限的几种服务，在 REST 风格下这些服务也可以称为资源。由 Arduino UNO 组成的 CoAP 服务器具有一个 hello 资源，通过 GET 方法可获得 hello 资源，hello 资源包含一个固定字符串内容“Hello CoAP!”。除了 hello 资源之外，CoAP 服务器还提供一个名为 light 的资源，该资源支持 GET 方法和 PUT 方法访问，通过 CoAP GET 方法可获得该资源内容，通过 CoAP PUT 方法可修改该资源的内容，结合 GET 和 PUT 方法，可以把虚拟资源的操作直接映射到真实资源中，例如此处的 light 资源对应一个真实的 LED，入门示例中真实的 LED 与 Arduino UNO 的 D8 脚相连。

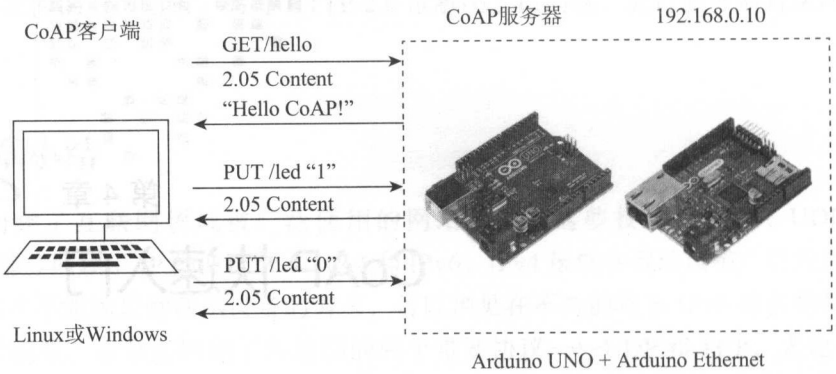


图 4-1 CoAP Arduino 示例

入门示例试图说明 CoAP 本身非常简单，而且几乎可以在任何硬件中运行。通过该入门示例可建立学习 CoAP 的信心。虽然本章的示例非常简单且容易掌握，但依然要求实验者掌握若干基本技能和技巧，这些技能和技巧包括：

- Arduino IDE 使用技能：如新建 Arduino UNO 工程、如何编写程序、如何下载固件等。
- Arduino 硬件操作技能：如如何给 Arduino 上电、如何连接以太网扩展模块、如何连接 LED 等。
- Git 工具使用技能：如如何通过 github 访问项目网址、如何复制代码、如何拉取最新代码等。

4.2 Copper 插件入门

Copper 是一款 Firefox 浏览器插件，毫无疑问 Copper 是最容易使用的 CoAP 客户端工具。由于 Copper 仅仅是 Firefox 浏览器的一个扩展插件，所以只要有 Firefox 浏览器便可使用 Copper 插件。无论在 Windows 平台还是 Linux 平台都可以随心所欲地使用 Copper 插件，并且使用步骤和体验细节都完全相同。

注意 Copper 仅仅是 Firefox 的一个扩展插件，并不能在 Chrome 或者 IE 浏览器中使用。

4.2.1 Copper 插件安装

Copper 插件的安装非常简单，首先需要保证计算机中已经安装了合适版本的 Firefox 浏览器。在 Firefox 浏览器的地址栏中输入 https://addons.mozilla.org/en-US/firefox/addon/copper-270430/，点击“Add to Firefox”，重启 Firefox 浏览器便可完成 Copper 插件安装。

安装 Copper 插件之后 Firefox 浏览器便可识别以“coap://”开头的 URI。如图 4-2 所示。

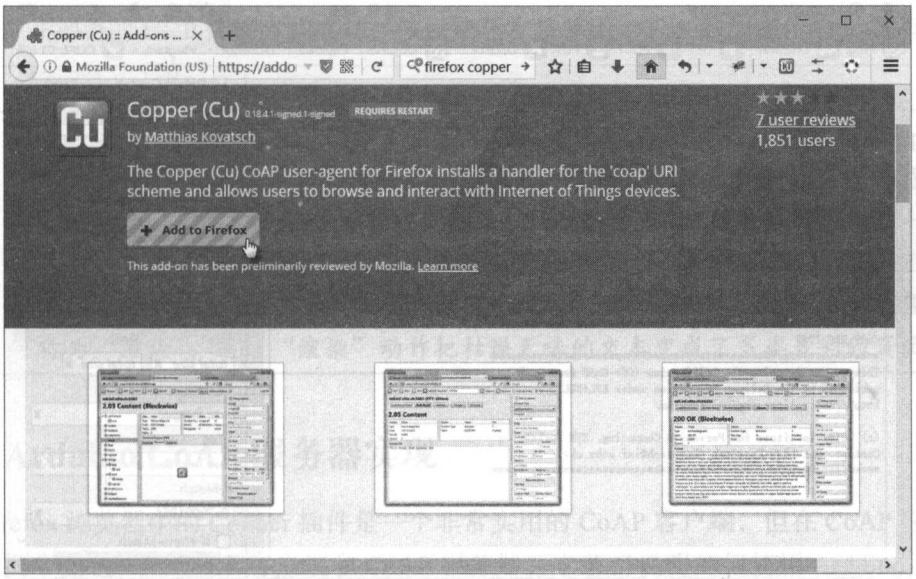


图 4-2　Firefox 浏览器中安装 Copper 插件

4.2.2　Copper 插件入门示例

虽然 Copper 插件安装过程非常简单，但 Copper 插件的功能却一点也不简单。CoAP 插件的具体使用细节请参考 8.2 节，此处仅通过入门示例说明 Copper 插件的使用方法。若 Copper 插件安装成功可使用 CoAP 测试服务器验证 Copper 插件的基本功能，入门示例中推荐两台 CoAP 测试服务器，它们分别是：

- 国外 CoAP 测试服务器：coap://californium.eclipse.org/
- 国内 CoAP 测试服务器：coap://wsncoap.org/

注意　CoAP 服务器的网址一般以 coap 开头，而常见的 Web 服务器的网址一般以 http 或 https 开头。国外 CoAP 服务器由 eclipse project 负责维护，而国内 CoAP 服务由本书作者负责维护，国内 CoAP 服务器部署于阿里云。一般来说，国内 CoAP 测试服务器较国外 CoAP 测试服务器更容易访问。

入门示例以国内服务器 coap://wsncoap.org/ 为例说明 Copper 插件的使用方法。在 Firefox 浏览器的地址栏中输入“coap://wsncoap.org/”，按下“Enter”键之后 Firefox 浏览器界面将发生明显变化，此时 Firefox 浏览器的外观如图 4-3 所示。在 Copper 插件的工具栏中点击“GET”按钮，此时 Copper 插件将使用 CoAP GET 方法向 wsncoap.org 服务器发送一次 CoAP 请求，wsncoap.org 服务器接收到请求之后将会返回 CoAP 响应。

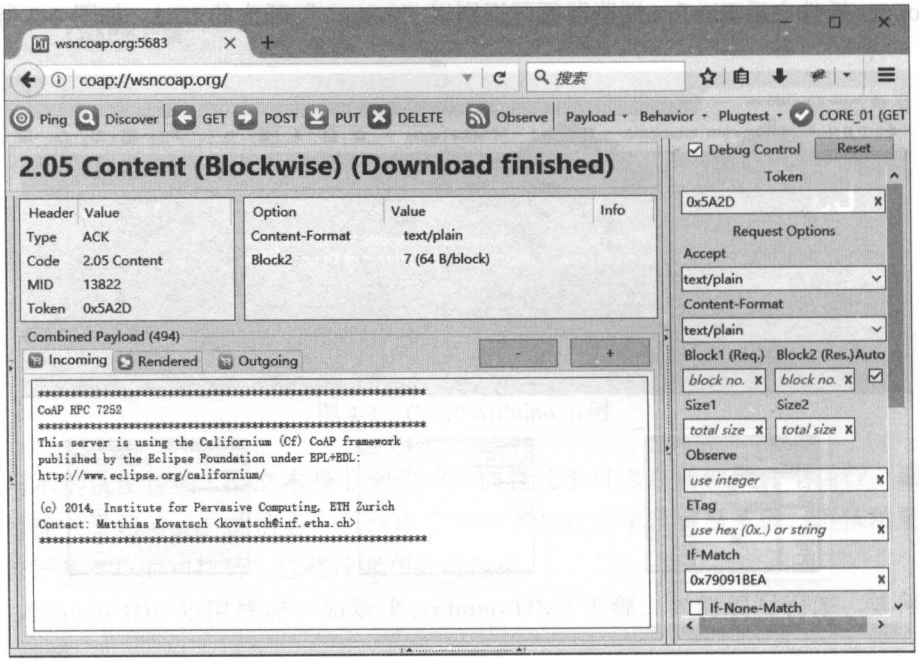


图 4-3 coap://wsncoap.org/

点击“GET”按钮之后，Copper 插件的 Incoming 选项卡中将不停地更新内容，若 Copper 插件完成接收 CoAP 响应，那么 Copper 插件的状态栏将显示“2.05 Content（Blockwise）（Download finished）”。在这个接收响应的过程中可以明显发现，CoAP 响应并不是以一个完整数据分包的形式返回至 CoAP 客户端，而是被分成了若干个数据分包依次返回至客户端。虽然 CoAP 服务器返回响应内容较少，但依然被 CoAP 服务器切割成多个数据分包返回至 CoAP 客户端。与 HTTP 不同，CoAP 主要为受限制低功耗设备服务，所以 CoAP 中包含了分组传输功能，默认情况下 Copper 默认数据分包的大小为 64 字节。相比于 HTTP，CoAP 的分包长度要小得多。

使用 CoAP GET 方法从 wsncoap.org 测试服务器获取的完整响应内容如下：

```
************************************************************
CoAP RFC 7252
************************************************************
This server is using the Californium (Cf) CoAP framework
published by the Eclipse Foundation under EPL+EDL:
http://www.eclipse.org/californium/

(c) 2014, Institute for Pervasive Computing, ETH Zurich
Contact: Matthias Kovatsch <kovatsch@inf.ethz.ch>
************************************************************
```

若仔细观察响应内容可以发现 wsncoap.org 服务器使用 Californium (Cf) CoAP frame-

work 实现 CoAP 服务器。Californium (Cf) CoAP framework 是一个非常完善的 CoAP 软件实现框架，可简称为 Cf 框架。Cf 框架既包括 CoAP 客户端实现也包括 CoAP 服务器端实现，除了常规非加密的 UDP 传输方式之外，该框架还支持基于 DTLS 的加密传输，Cf 框架的具体使用方法详见 7.5 节。

注意　Copper 插件通过图文并茂的方式展现 CoAP 请求和响应结果，再加上 Copper 插件在浏览器中运行，使得首次使用 CoAP 的用户觉得 CoAP 也和 HTTP 一样存在“页面”的概念。其实 CoAP 并没有“页面”，它仅仅是一个应用层协议。从另一个角度来说 HTTP 也没有“页面”，HTTP 只是从 Web 服务器中获取了相应的 HTML 文件，浏览器把获取的 HTML 文件中每个标记重新展现至终端用户，这种重新展现的过程可称为“渲染”，通过“渲染”动作把枯燥无味的文本变成了多姿多彩的页面。

4.3　Arduino CoAP 服务器实现

Firefox 浏览器中的 Copper 插件是一个非常实用的 CoAP 客户端，但在 CoAP 应用中总是存在两个角色——CoAP 客户端和 CoAP 服务器，本节将说明如何使用 Arduino UNO 和 Arduino Ethernet Shield 以太网扩展板实现一个非常简洁的 CoAP 服务器。

4.3.1　获取示例

在 Arduino UNO CoAP 服务器部分的实现代码位于本书代码仓库 the_beginning_of_coap 中，可通过 Git 工具复制本书提供的示例代码。

```
# 新建一个名为repo的文件夹（repo 是repository仓库的简称）
mkdir -P repo
# 复制代码仓库
git clone https://github.com/xukai871105/the_beginning_of_coap.git
# 进入示例代码目录
cd the_beginning_of_coap
```

Arduino CoAP 服务器实现的示例代码位于 first_demo/microcoap 目录中。该目录中包括 3 个重要文件，即 microcoap.ino、coap.c 和 coap.h，其中：

- microcoap.ino 为 Arduino UNO 的工程文件，该文件实现串口初始化、网络初始化、接收 CoAP 请求、处理 CoAP 请求和返回 CoAP 响应等功能，可在 microcoap.ino 文件中增加各种 CoAP 路由，这些路由可称为 endpoint（端点），又可称为资源。入门示例中包含两个 endpoint（端点）：一个 endpoint 名为“hello”，另一个 endpoint 名为“led”。
- coap.c 和 coap.h 为 CoAP 的实现代码，该部分代码实现了 CoAP 首部解析和填充、选项解析和填充、负载分离和填充等功能。一般情况下 coap.c 和 coap.h 并不需要修改。coap.c 和 coap.h 仅实现 CoAP 中很多基础功能，但它并不包括 CoAP 重传和 CoAP 分组传输等其他功能。

4.3.2 示例说明

microcoap.ino 可分为初始化、CoAP 数据处理、endpoints 列表、hello 资源和 light 资源等部分，各部分的详细说明如下。

1. 初始化

在 Arduino 应用程序中总有一个 setup 函数和一个 loop 函数，setup 函数主要用于完成设备初始化工作，在入门示例中，setup 函数主要实现 led 初始化、串口初始化和网络设备初始化等工作，具体实现代码如下。

代码清单4-1 初始化部分代码

```
#include <SPI.h>
#include <Ethernet.h>
#include <EthernetUdp.h>

static int led = 8;
// 自定义网卡地址
byte mac[] = {0x00, 0xAA, 0xBB, 0xCC, 0xDE, 0x02};
// Arduino主机地址，请根据实际网络情况修改
IPAddress ip(192, 168, 0, 10);
IPAddress gateway(192, 168, 0, 1);
IPAddress subnet(255, 255, 255, 0);

EthernetUDP udp;
uint8_t packetbuf[256];

void setup()
{
    int i;
    pinMode(led, OUTPUT);
    Serial.begin(9600);

Ethernet.begin(mac, ip);
// 打印本机IPv4地址
    Serial.print("My IP address: ");
    for (i = 0; i < 4; i++)
    {
        Serial.print(Ethernet.localIP()[i], DEC);
        Serial.print(".");
    }
Serial.println();
// 侦听5683端口的UDP输入数据
    udp.begin(5683);
}
```

在 setup 函数中：

- pinMode（led, OUTPUT）初始化 LED 为输出模式，入门示例中 LED 与 Arduino 的 D8 脚相连。

- Serial.begin（9600）设置串口波特率，此处的波特率为 9600bit/s。借助 Arduino 的串口功能打印程序运行过程中的调试信息，这些调试信息包括 Arduino 本机 IPv4 地址、其他主机发送至 Arduino 的 UDP 数据包等。
- Ethernet.begin（mac, ip）设置 W5100 网卡的 MAC 地址和 IPv4 地址。
- udp.begin（5683）绑定 UDP 5683 端口号，通过 5683 端口接收 CoAP 请求。

2. CoAP 数据处理

CoAP 数据处理部分位于 loop 函数中，CoAP 数据处理部分从 Arduino UDP 服务器演变而来。通过 udp.parsePacket 函数可获取当前 UDP 数据包字节长度，如果 UDP 数据包长度大于 0 说明 Arduino 接收到其他主机发送而来的 CoAP 请求；通过 coap_parse 函数验证并解析该 CoAP 请求；若验证通过将遍历 endpoints 数组，endpoints 数组包含 hello 资源处理函数和 light 资源处理函数，若 CoAP 请求中的 URI 与相应 endpoint 的 URI 匹配，那么 coap_handle_req 将会通过函数指针的方式调用资源处理函数；若 CoAP 请求执行完成，将通过 coap_build 构造 CoAP 响应；最后通过 udp_send 函数向 CoAP 客户端返回响应内容。CoAP 请求的处理流程如图 4-4 所示。

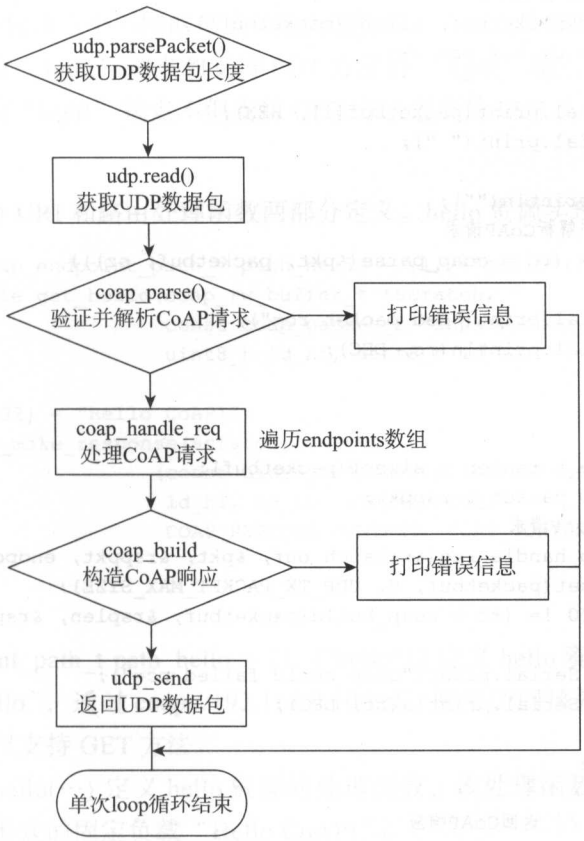


图 4-4　CoAP 请求处理流程

CoAP 请求处理部分代码如下所示。

代码清单4-2 CoAP请求处理

```
void udp_send(const uint8_t *buf, int buflen)
{
    udp.beginPacket(udp.remoteIP(), udp.remotePort());
    while(buflen--)
        udp.write(*buf++);
    udp.endPacket();
}

void loop()
{
    int sz;
    int rc;
    coap_packet_t pkt;
    int i;

    if ((sz = udp.parsePacket()) > 0)
    {
        // 读取UDP请求内容
        udp.read(packetbuf, sizeof(packetbuf));
        for (i = 0; i < sz; i++)
        {
            Serial.print(packetbuf[i], HEX);
            Serial.print(" ");
        }
        Serial.println("");
        // 验证并解析CoAP请求
        if (0 != (rc = coap_parse(&pkt, packetbuf, sz)))
        {
            Serial.print("Bad packet rc=");
            Serial.println(rc, DEC);
        }
        else
        {
            size_t rsplen = sizeof(packetbuf);
            coap_packet_t rsppkt;
        // 处理CoAP请求
            coap_handle_req(&scratch_buf, &pkt, &rsppkt, endpoints);
            memset(packetbuf, 0, UDP_TX_PACKET_MAX_SIZE);
            if (0 != (rc = coap_build(packetbuf, &rsplen, &rsppkt)))
            {
                Serial.print("coap_build failed rc=");
                Serial.println(rc, DEC);
            }
            else
            {
                // 返回CoAP响应
```

```
                udp_send(packetbuf, rsplen);
            }
        }
    }
}
```

3. endpoints 列表

程序进行 CoAP 请求处理时将遍历 endpoints 列表，endpoints 列表可理解为一组端点集合（也可以理解为路由集合或资源集合），每个端点表示一个 CoAP 请求处理的最小“单元”。在 microcoap.ino 中，每一个端点由请求方法、请求处理函数、请求 URI 和媒体类型组成，endpoints 列表的具体定义如下。

```
coap_endpoint_t endpoints[] =
{
    {COAP_METHOD_GET, handle_get_hello, &path_hello, "ct=0"},
    {COAP_METHOD_GET, handle_get_light, &path_light, "ct=0"},
    {COAP_METHOD_PUT, handle_put_light, &path_light, NULL},
    {(coap_method_t)0, NULL, NULL, NULL}
};
```

microcoap.ino 中包含三个不同的端点（路由或资源）——支持 GET 方法的“hello”端点、支持 GET 方法的“light”端点和支持 PUT 方法的“light”端点。此处“hello”端点仅支持 GET 方法，而“light”端点不但支持 GET 方法还支持 PUT 方法。

4. hello 资源

hello 资源需要由 URI 和路由处理函数两部分定义。hello 资源实现代码如下：

```
static const coap_endpoint_path_t path_hello = {1, {"hello"}};
static int handle_get_hello(coap_rw_buffer_t *scratch,
                            const coap_packet_t *inpkt, coap_packet_t *outpkt,
                            uint8_t id_hi, uint8_t id_lo)
{
    char hello[32] = "Hello CoAP!";
    return coap_make_response(scratch, outpkt,
                              (const uint8_t *)&hello, strlen(hello),
                              id_hi, id_lo, &inpkt->tok,
                              COAP_RSPCODE_CONTENT, COAP_CONTENTTYPE_TEXT_PLAIN);
}
```

在上述代码中：

- coap_endpoint_path_t path_hello = {1, {"hello"}} 定义 hello 资源的 URI，该资源的 URI 为“hello”，通过 coap://192.168.0.10/hello 便可访问该资源。Arduino 入门示例中该资源仅支持 GET 方法。
- handle_get_hello(…) 定义 hello 资源的处理函数，该处理函数将会向 CoAP 客户端返回字符串形式的固定负载“Hello CoAP!”。

5. light 资源

light 资源也由 URI 和路由处理函数两部分组成。相比于 hello 资源，light 资源既支持 GET 方法也支持 PUT 方法。

```
static char light = '0';
static int led = 8;
static const coap_endpoint_path_t path_light = {1, {"light"}};
static int handle_get_light(coap_rw_buffer_t *scratch,
                            const coap_packet_t *inpkt, coap_packet_t *outpkt,
                            uint8_t id_hi, uint8_t id_lo)
{
    return coap_make_response(scratch, outpkt,
                              (const uint8_t *)&light, 1,
                              id_hi, id_lo, &inpkt->tok,
                              COAP_RSPCODE_CONTENT, COAP_CONTENTTYPE_TEXT_PLAIN);
}

static int handle_put_light(coap_rw_buffer_t *scratch,
                            const coap_packet_t *inpkt, coap_packet_t *outpkt,
                            uint8_t id_hi, uint8_t id_lo)
{
    // 若CoAP负载的首个字符为1，说明CoAP客户端试图打开LED
    if (inpkt->payload.p[0] == '1')
    {
        light = '1';
        digitalWrite(led, HIGH);
        Serial.println("ON");
    }
    else
    {
        light = '0';
        digitalWrite(led, LOW);
        Serial.println("OFF");
    }
    return coap_make_response(scratch, outpkt,
                              (const uint8_t *)&light, 1,
                              id_hi, id_lo, &inpkt->tok,
                              COAP_RSPCODE_CHANGED,
                              COAP_CONTENTTYPE_TEXT_PLAIN);
}
```

在上述代码中：

- coap_endpoint_path_t path_light = {1, {"light"}} 定义该资源的 URI，该资源的 URI 为“light”，可通过 coap://192.168.0.10/light 便可访问该资源，与 hello 资源不同，该资源既支持 GET 方法也支持 PUT 方法，也就是说可通过 PUT 方法改变资源的状态，即通过 PUT 方法修改 LED 点亮或熄灭。
- handle_get_light() 即 light 资源处理函数之一，该处理函数支持 GET 方法，将返回

此时 LED 的状态，LED 的状态通过一个名为 light 的字符变量进行保存，当 light 的值为‘0’时，LED 处于熄灭状态；light 的值为‘1’时，LED 处于点亮状态。

- handle_put_light() 即 light 资源处理函数之一，该处理函数支持 PUT 方法。若请求负载的值为‘1’时，通过 digitalWrite（led, HIGH）点亮 LED；若请求负载的值不为‘1’时，通过 digitalWrite（led, LOW）熄灭 LED。

4.3.3　动手测试

通过以上代码分析应该对入门示例的运行流程有一个大体的了解，下面再通过动手测试环节加深对上述代码和运行流程的理解。

1. 网络参数修改

开始运行代码之前建议根据网络的实际情况修改入门示例中的相关参数，示例代码中与网络有关的参数一共有三项：

```
IPAddress ip(192, 168, 0, 10);
IPAddress gateway(192, 168, 0, 1);
IPAddress subnet(255, 255, 255, 0);
```

通过 IPAddress ip（192, 168, 0, 10）指定 Arduino 的 IPv4 地址为 192.168.0.10，该 IPv4 地址为一个局域网 IPv4 地址。在大多情况下 PC 的 IPv4 地址均由路由器分配获得，但在入门示例中 Arduino 使用固定 IPv4 地址，一定要确认 Arduino 的 IPv4 地址处于路由器所指定的网段中；IPAddress gateway（192, 168, 0, 1）指定了路由器的 IPv4 地址，IPAddress subnet（255, 255, 255, 0）指定子网掩码。如果不清楚本地局域网信息，可在 Windows 主机中通过控制台输入 ipconfig 命令查询。

例如，路由器的 IP 地址为 192.168.1.1，子网掩码为 255.255.255.0，测试使用 Windows 主机的 IPv4 地址为 192.168.1.101，可设置 Arduino 的 IPv4 地址为 192.168.1.108，修改之后的网络参数如下：

```
IPAddress ip(192, 168, 1, 108);
IPAddress gateway(192, 168, 1, 1);
IPAddress subnet(255, 255, 255, 0);
```

完成了网络参数的设置之后，那么便可动手进行测试工作，相比于枯燥的文档说明，调试过程更容易积累 CoAP 的使用经验。在 Arduino 中运行示例工程，先把固件下载至 Arduino UNO 目标板中，操作过程如图 4-5 所示。

2. 连接 CoAP 服务器

完成固件下载之后先使用 ping 命令查看 Arduino UNO 是否可达，若未收到 Arduino 的响应需检查 Arduino 相关的网络参数，并及时修改 microcoap.ino 中的具体设置。本例中 Arduino 的 IP 地址为 192.168.0.10，固件下载完成之后可顺利 ping 通 Arduino 设备，若获得如图 4-6 所示的相似结果，则说明 Arduino UNO 和 Arduino 网络扩展卡工作良好。

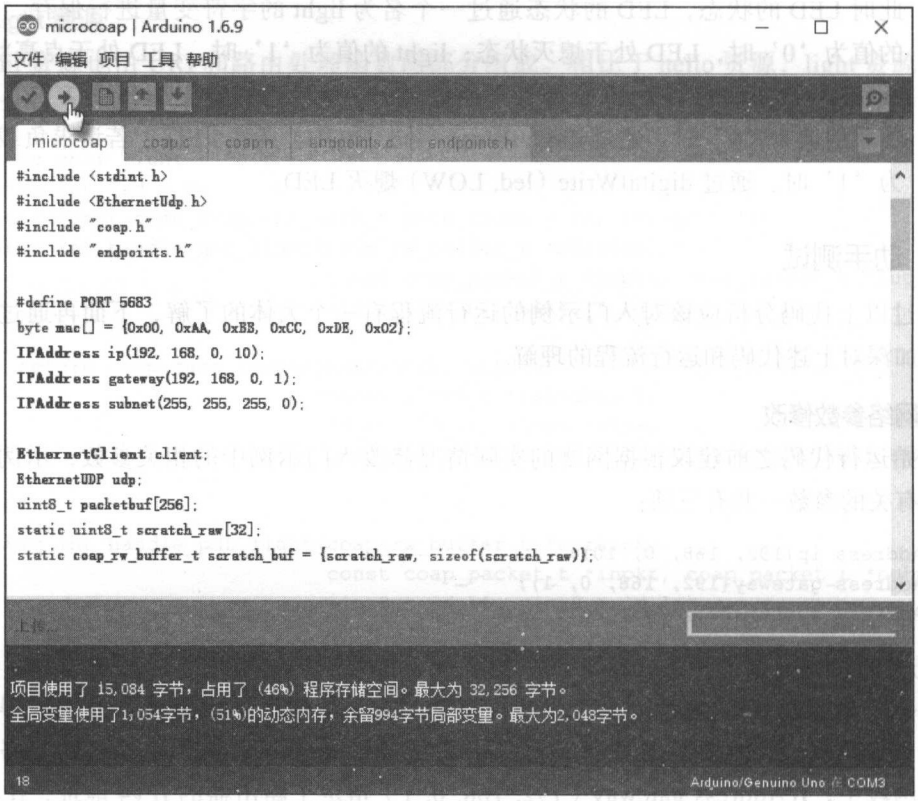


图 4-5　下载入门示例固件至 Arduino 目标板中

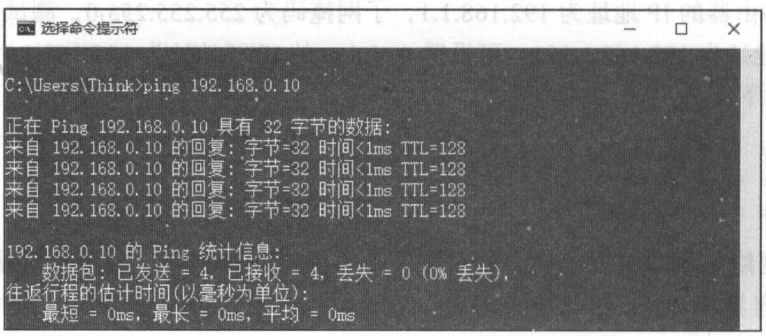


图 4-6　连接 CoAP 服务器

3. 获取 hello 资源

验证了网络连通性之后可使用 Firefox 浏览器中的 Copper 插件获取 hello 资源，具体操作过程如图 4-7 所示。

1）打开 Firefox 浏览器，在地址栏中输入 coap://192.168.0.8:5683/hello，按下回车键，

浏览器界面出现明显变化。

2）点击工具栏中的 GET 按钮，通过浏览器向 Arduino UNO 发送一次 CoAP GET 请求。

3）点击 GET 按钮之后便可在负载区域 Incoming 选项卡中观察到“Hello CoAP！”，此时 CoAP 响应码为“2.05 Content”。

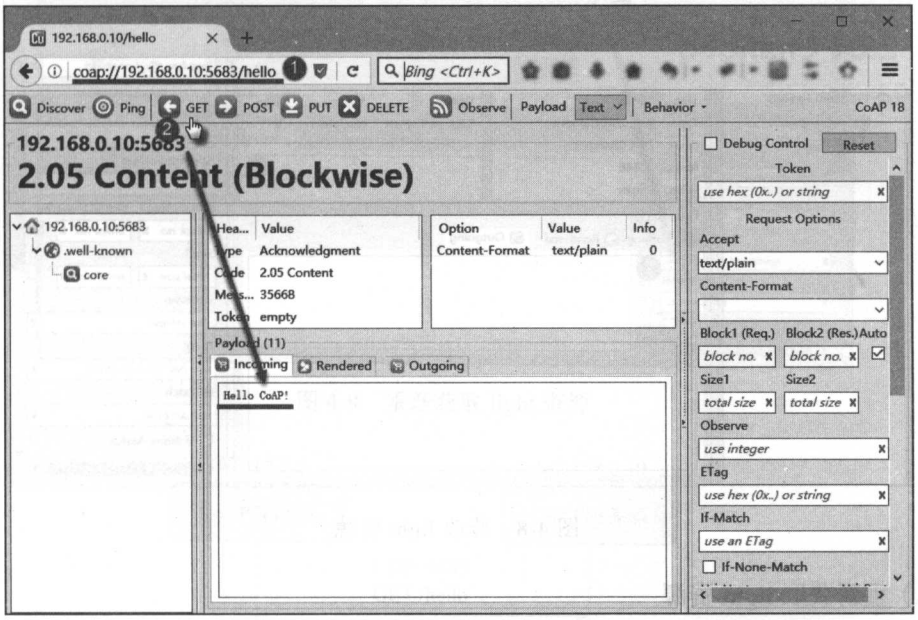


图 4-7　使用 Copper 插件获取 hello 资源

GET 方法是 CoAP 中最常用的方法之一，由于 hello 资源本身的限制每次通过 GET 方法访问资源后只能获得一个固定的响应——“Hello CoAP！”。下面通过一个可变化的过程说明如何通过 CoAP 控制设备。

4. 访问与修改 light 资源

与 hello 资源不同，light 资源可以被访问也可以被修改。验证 light 资源可分为两步，第一步通过 PUT 方法修改资源，负载内容设置为“1”，PUT 请求成功发送之后 LED 将处于点亮状态；第二步通过 GET 方法重新获取该资源，查看获取的负载是否已经变为“1”。

（1）使用 PUT 方法修改资源

通过 PUT 方法可控制 LED 状态使其点亮或熄灭，若点亮 LED 可把 PUT 请求负载设置为“1”，具体操作过程如图 4-8 所示。

1）打开 Firefox 浏览器，在地址栏中输入 coap://192.168.0.10:5683/light。

2）在负载区域的 outgoing 选项卡中输入“1”。

3）点击“PUT”按钮，向 Arduino 发送一次 CoAP PUT 请求。

4）若 Arduino 正确处理 PUT 请求，将会向 Copper 插件返回响应，此时响应码为

“2.04 Changed”。

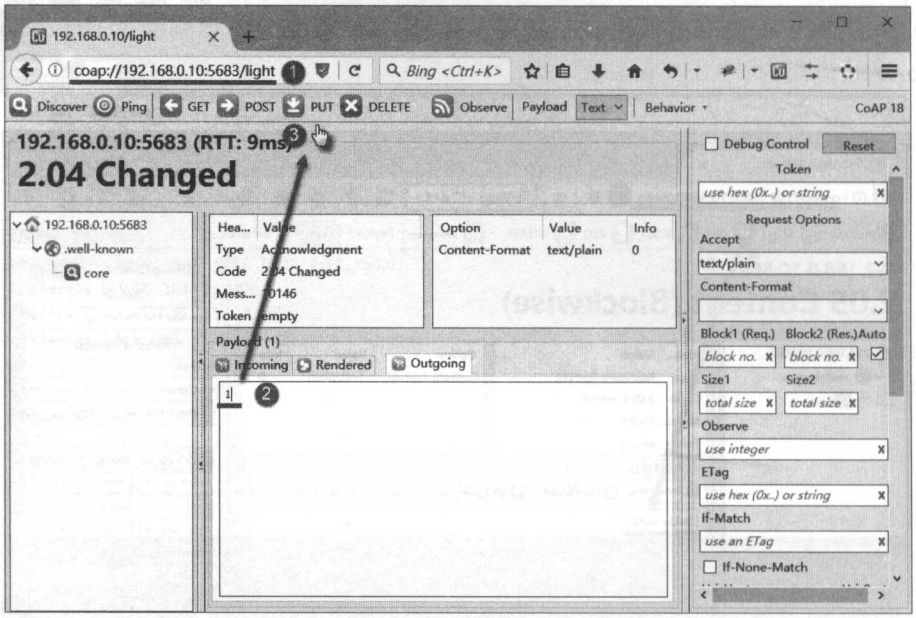


图 4-8　修改 light 资源

（2）重新获取 light 资源

通过上一步操作 LED 将处于点亮状态，light 资源也已经发生改变，通过 GET 方法可以重新获取该资源以验证资源是否被正确修改，验证过程如图 4-9 所示。

1）打开 Firefox 浏览器，在地址栏中输入 coap://192.168.0.10:5683/light。

2）点击工具栏中的“GET”按钮，向 Arduino 发送 GET 请求。

3）若 Arduino 正确处理 CoAP 请求，可在负载区域 Incoming 选项卡中观察到“1”，此时 CoAP 响应的响应码为“2.05 Content”。

4.3.4　着手分析

最后再通过 Wireshark 抓取网络数据尝试分析 CoAP 的具体细节。通过前面几步操作已经对 CoAP 工作流程有一个大致的了解，那么通过 Wireshark 工具可对 CoAP 本身有一个更全面的了解。当然此时不熟悉 CoAP 的细节也没有关系，第 5 章将会对此进行更为详细的分析。

打开 Wireshark，选择合适的网卡侦听网络分组数据，如选择有线网卡或无线网卡；在 Wireshark 过滤栏中输入“ coap”。再次使用 Copper 插件访问 Arduino UNO 中的 hello 资源。在 Wireshark 中可以“截获”两个 UDP 数据包，其中一个 UDP 数据包代表 CoAP 请求，另一个 UDP 数据包代表 CoAP 响应。两个 UDP 数据包数据流向和大致内容如图 4-10 所示。

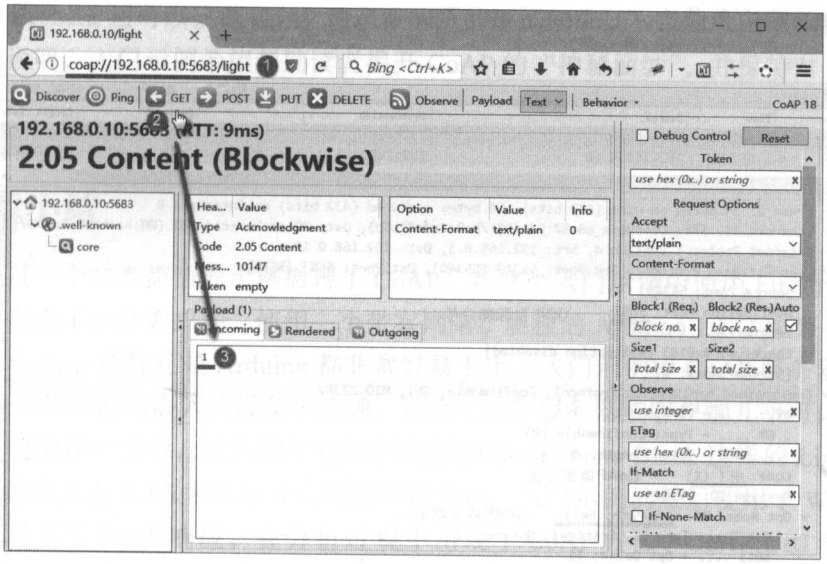


图 4-9　重新获取 light 资源

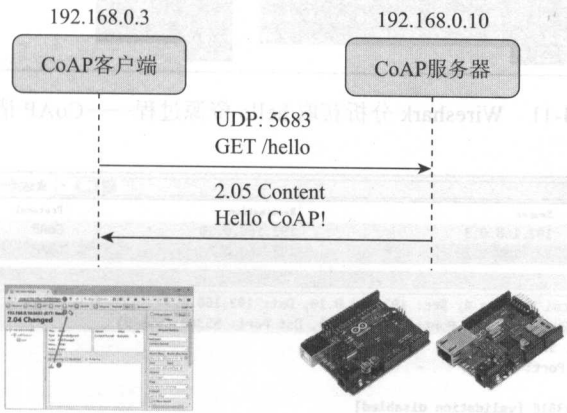


图 4-10　通过 Wireshark 获取 CoAP 请求响应

1. CoAP 请求分析

在 CoAP 请求中可以观察到如图 4-11 所示的类似内容，这些内容包括：

- UDP 目标端口号为 5683，该端口为 CoAP 协议的知名端口（默认端口）。
- CoAP 请求方法为 GET。
- CoAP 请求路由为“hello”。

2. CoAP 响应分析

在 CoAP 响应中可以观察到如图 4-12 所示的类似内容，这些内容包括：

- CoAP 响应码为“2.05 Content”。

❑ CoAP 响应负载为“Hello CoAP！”。

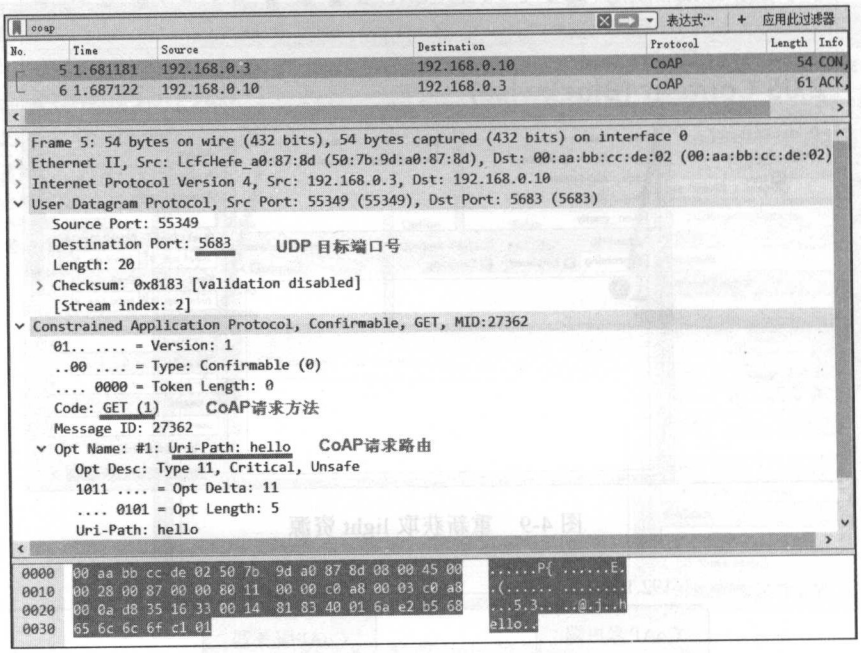


图 4-11　Wireshark 分析获取 hello 资源过程——CoAP 请求

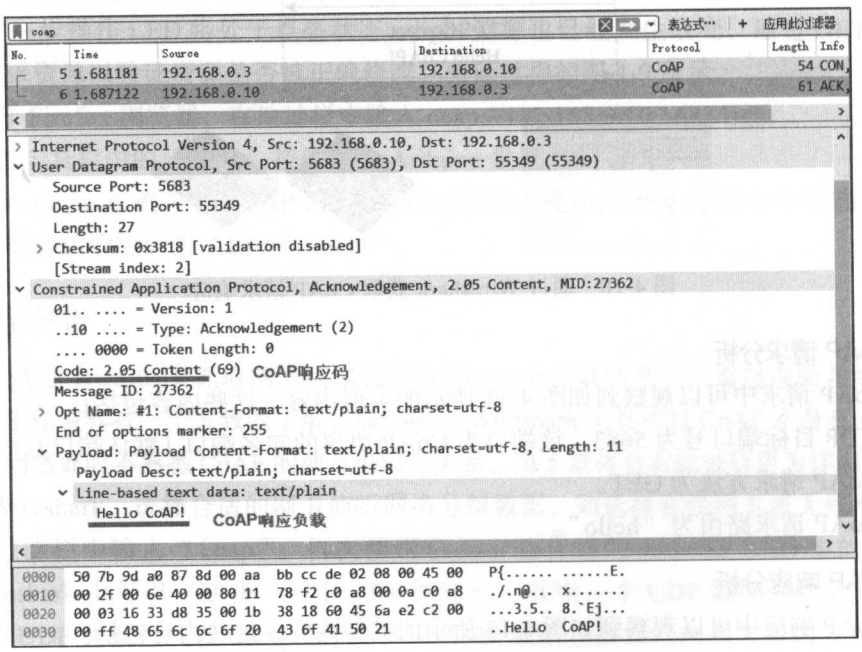
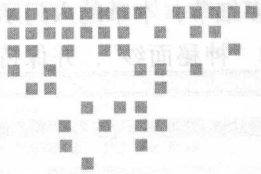


图 4-12　Wireshark 分析获取 hello 资源过程——CoAP 响应

通过 Wireshark 还可以获得 CoAP 请求或响应的更多细节，本章的入门示例将不对这些细节做详细说明，入门示例希望帮助读者揭开 CoAP 的“神秘面纱”，并保持对 CoAP 的好奇心与信心。

4.4　本章小结

本章通过一个非常简洁的示例展现了 CoAP，在这个入门示例中使用 Firefox 浏览器的 Copper 插件作为 CoAP 客户端，使用一个带有网络扩展板的 Arduino UNO 作为 CoAP 服务器。无论是 Copper 插件还是 Arduino 都非常容易上手。入门示例中 CoAP 服务器在一个内存只有 2KB 的受限制的嵌入式设备中实现，那么 CoAP 服务器或客户端几乎可以在任何具有联网能力的硬件中实现，这些硬件包括树莓派和 Beaglebone Black 这样的 Linux 主机设备，也包括那些具有 IEEE 802.15.4 无线射频的 SoC。入门示例的最后使用了 Wireshark 分析了 CoAP 请求与 CoAP 响应，在分析过程中出现了 CoAP 方法、CoAP URI 和 CoAP 响应码等名词，通过后面几章的学习读者可以很轻松地掌握这些概念，让我们继续 CoAP 之旅吧。



Chapter 3 第 5 章

CoAP 核心

5.1 本章主要内容

本章将学习 CoAP 的核心部分内容，CoAP 核心部分由《 RFC 7252—The Constrained Application Protocol (CoAP) [⊖]》定义，本章以该 RFC 文档为主线，包括以下主要内容：

- CoAP 首部分析：版本编号、报文类型、标签长度指示、准则、报文序号、标签、选项、分隔符和负载。
- CoAP 工作模式说明：CON、NON、ACK 和 RST。
- CoAP 重传机制分析：CoAP 请求丢失处理、CoAP 响应丢失处理、最大重传次数、最大传输耗时、最大等待时间。
- CoAP 方法说明：GET 方法、POST 方法、PUT 方法和 DELETE 方法。
- CoAP 响应码说明：正确响应、客户端错误、服务器错误。
- CoAP 选项详细分析：选项格式、URI 选项、Content-Format 选项、Accept 选项、Etag 选项、If-Match 选项、If-None-Match 选项。
- CoAP 媒体类型说明：link-format 类型、文本类型、二进制类型、JSON 类型。

5.2 CoAP 首部

与 UDP 和 TCP 类似，CoAP 的学习也应从 CoAP 首部开始。与 HTTP 1.X 协议不同，CoAP 是一个完整的二进制应用层协议，CoAP 首部包括版本编号 Ver、报文类型 T、标签

⊖ https://datatracker.ietf.org/doc/rfc7252/。

长度指示 TKL、准则 Code、报文序号 Message ID、标签 Token、选项 Options、分隔符 0xFF 和负载 Payload 等几个部分。

其中版本编号 Ver、报文类型 T、标签长度指示 TKL、准则 Code 和报文序号 Message ID 为必要部分，也就是说这几部分一定会出现在 CoAP 请求或响应中。而标签 Token、选项 Options、分隔符 0xFF 和负载 Payload 为非必要部分，这些部分均为可选部分。CoAP 首部结构如图 5-1 所示。

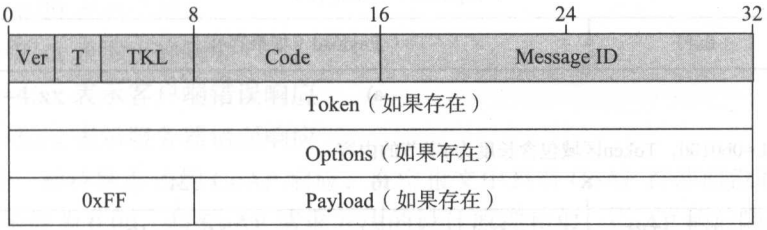


图 5-1　CoAP 首部结构

5.2.1　版本编号 Ver

CoAP 版本编号区域占 2 位。在 RFC 7252 规范中该区域必须为固定值 0b01。

5.2.2　报文类型 T

CoAP 报文类型区域占 2 位。CoAP 中共定义了 4 种不同的报文类型，分别是：

- Confirmable：需要被确认的报文，简称 CON 报文，此时 T=0b00。
- Non-Confirmable：不需要被确认的报文，简称 NON 报文，此时 T=0b01。
- Acknowledgement：应答报文，简称 ACK 报文，此时 T=0b10。
- Reset：复位报文，简称 RST 报文，此时 T=0b11。

在 HTTP 中，一个 HTTP 请求对应一个 HTTP 响应；但在 CoAP 中，如果 CoAP 客户端发送 NON 类型的 CoAP 请求，那么 CoAP 服务器可选择不返回 CoAP 响应。换句话说，CoAP 客户端并不关心请求是否到达 CoAP 服务器。NON 类型报文是 CoAP 中的一个“特色”，通过这种设计允许设备犯“错”。

5.2.3　标签长度指示 TKL

CoAP 标签长度指示 TKL 占 4 位，该区域用于指示 CoAP 标签区域的具体长度。由于 CoAP 是一个二进制协议，对于非固定长度的区域都需要长度指示。CoAP 报文中可以包含 CoAP 标签也可以省略 CoAP 标签。若 CoAP 报文中省略 CoAP 标签，那么此时的 TKL=0b0000；若 CoAP 报文中包含 CoAP 标签，那么 TKL 的取值可以为 0b0001(1)、0b0010(2) 或 0b0100(4)。图 5-2a 中 TKL=0b0001(1) 时，Token 区域包含长度为 1 字节的内容；

图 5-2b 中 TKL=0b0100(4) 时，Token 区域包含长度为 4 字节的内容。

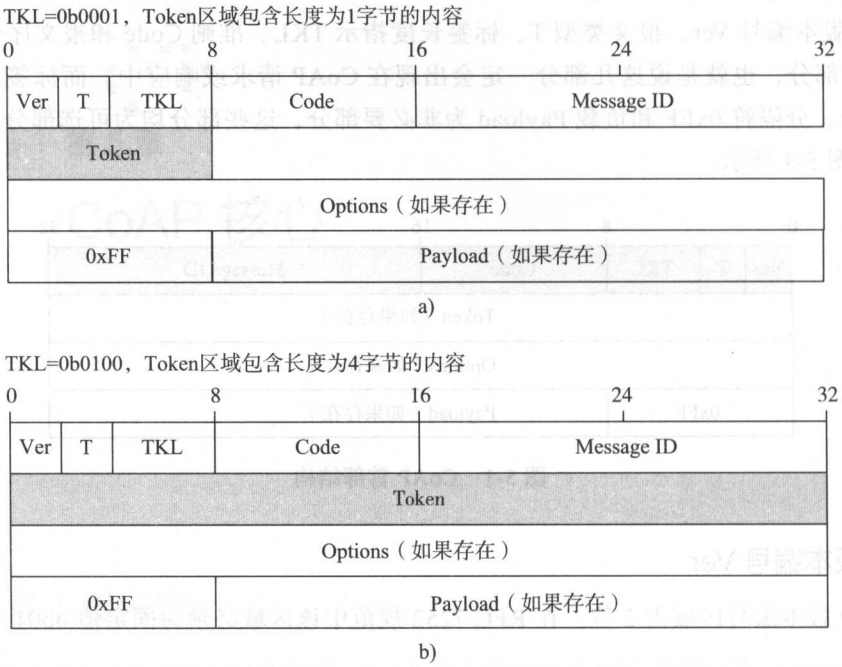


图 5-2　标签长度指示 TKL 示例

5.2.4　准则 Code

CoAP 准则占 1 字节（8 位）。虽然该区域仅占 8 位，但该区域在 CoAP 请求和响应报文中却包含了大量有用信息。Code 部分分为高 3 位 Class 部分和低 5 位 Detail 部分。为了更方便地描述和表达，Code 部分采用 c.dd 的形式描述，其中 c 的取值范围为 0～7，dd 的取值范围为 0～31。“c”部分和“dd”部分均采用十进制形式描述。当 c 等于 0 时表示 CoAP 请求，当 c 不等于 0 时表示 CoAP 响应。

1. CoAP 请求

在 CoAP 请求报文中 Code 区域用于指示 CoAP 请求方法，CoAP 请求方法和 HTTP 请求方法非常相似，CoAP 中共有 4 种不同的请求方法——GET 方法、POST 方法、PUT 方法和 DELETE 方法。

- Code=0.01 表示 GET 方法
- Code=0.02 表示 POST 方法
- Code=0.03 表示 PUT 方法
- Code=0.04 表示 DELETE 方法

与 HTTP 不同，CoAP 协议使用二进制方式表示请求方法，无论如何这些请求方法仅占

1 字节，而在 HTTP 中请求方法采用字符串形式表达，“GET”占 3 字节，“POST”占 4 字节。CoAP 通过这样的设计方式既缩短了 CoAP 首部长度又与 HTTP 协议保持相同的请求语义。

2. CoAP 响应

CoAP 响应报文中 Code 区域用于指示 CoAP 响应状态。CoAP 响应码和 HTTP 中的状态码非常相似，CoAP 中定义了 4 种不同类型的响应报文——空报文、正确响应、客户端错误响应和服务器错误响应。

- Code=0.00 表示空报文
- Code=2.xx 表示正确响应
- Code=4.xx 表示客户端错误响应
- Code=5.xx 表示服务器错误响应

空报文是一种特殊形式的 CoAP 响应，在空报文中只有 CoAP 首部而没有 CoAP 负载，且 Code 区域始终为 0.00。与 CoAP 请求方法的设计原理相似，CoAP 响应码也与 HTTP 状态码保持相似的语义，如 HTTP 中 2XX 表示正确响应，而 CoAP 中 2.xx 也表示正确响应。在 CoAP 中，2.05 Content 与 HTTP 200 OK 的含义几乎相同，但是 CoAP 中表示成功仅占 1 字节，HTTP 中“200 OK”却占 6 字节。

5.2.5　报文序号 Message ID

CoAP 报文序号占 2 字节，并采用大端格式描述。由于 CoAP 采用 UDP 作为传输层协议，UDP 不能保证 CoAP 报文的到达顺序。如果没有报文序号，那么无论客户端还是服务器都无法建立报文之间准确的一一对应关系。CoAP 中规定，一组对应的 CoAP 请求和 CoAP 响应必须使用相同的 Message ID。

5.2.6　标签 Token

标签是一个长度可变的区域，该区域的长度由 TKL 定义，一般为 1 字节、2 字节或 4 字节。在 CoAP 中，标签可以理解为另一种形式的报文序号。CoAP 中定义了两种不同形式的请求 / 响应工作模式：一种为携带模式；另一种为分离模式。标签在分离模式中发挥重要的作用，但在携带模式中往往可以省略。

5.2.7　选项 Options

CoAP 请求或响应中可携带一组或多组 CoAP 选项，CoAP 选项和 HTTP 中的通用首部字段、请求首部字段、响应首部字段和实体首部字段功能相似。CoAP 选项是 CoAP 核心协议中较为复杂的部分，但选项部分也给 CoAP 的应用带来了诸多灵活性。CoAP 选项包括 Uri-Host、Uri-Port、Uri-Path、Uri-Query、Content-Format、Accept、Etag、If-Match 和 If-None-Match 等部分。

5.2.8 分隔符 0xFF

CoAP 首部和 CoAP 负载之间使用固定分隔符 0xFF，占 1 字节。在 HTTP 中，HTTP 首部和负载之间也有一个显式的分隔标记——空行（回车与换行，共占 2 字节）。CoAP 中也保留了类似的分隔符，在 CoAP 首部的其他部分并没有首部长度的指示，且首部长度也不是一个固定值，那么就需要 0xFF 这样的固定分隔符区分 CoAP 首部和 CoAP 负载。

在一个具有 CoAP 负载的报文中，固定分隔符将会出现在标签 Token 或者 CoAP 选项 Options 之后。如果固定分隔符出现在标签 Token 之后，由于 CoAP 首部中有标签长度指示，即使标签 Token 区域中最后一字节为 0xFF 也不会影响报文的解析；如果固定分隔符出现在选项 Options 之后，由于每个单独的选项也具有长度指示，那么即使最后一个选项以 0xFF 结尾也不会影响报文解析。在上述两种极端情况下，将在 CoAP 报文中连续出现两个 0xFF，但不会影响报文的相关解析。

在图 5-3 中，Token 区域的最后一个字符为 0xFF，在 CoAP 报文中连续出现了两个 0xFF，但由于 TKL 指示 Token 区域的长度为 4 字节，通过 TKL 就可以正确识别出哪一个 0xFF 属于 Token 区域，哪一个 0xFF 属于 CoAP 首部与 CoAP 负载之间的分隔符。

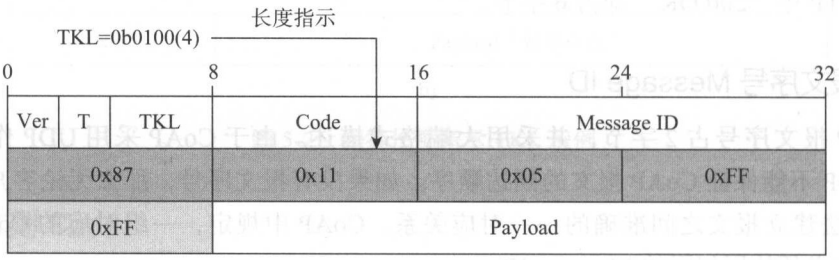


图 5-3　连续出现两次 0xFF 的极端情况

5.2.9 负载 Payload

CoAP 负载包含与具体应用直接相关的内容。CoAP 负载包含多种不同的媒体类型，包括二进制负载、文本负载、XML 负载、JSON 负载、CBOR 负载等。CoAP 所支持的媒体类型中并不包括 HTML 类型，换句话说 CoAP 并不包含与网页相关的内容，这点与 HTTP 存在很大的区别。

5.3 CoAP 工作模式

CoAP 虽然参考了 HTTP 的设计思路，但也根据受资源限制设备的具体情况改良了诸多设计细节，同时增加了很多实用功能。本节分为三部分内容——逻辑分层结构、报文类型和请求 / 响应模式。

5.3.1　逻辑分层结构

CoAP 的数据交互方式和 HTTP 的请求 / 响应工作方式非常相似。CoAP 请求一般由客户端发起，服务器根据客户端请求中的 URI 定位资源在服务器中的具体位置，通过客户端请求中的请求方法确定如何操作该资源，如读取资源、创建资源、修改资源或者删除资源等。CoAP 服务器处理请求之后将返回一个 CoAP 响应，CoAP 响应中包含响应码，也有可能包含响应负载。

与 HTTP 采用 TCP 作为传输层不同，CoAP 使用 UDP 作为传输层协议。UDP 并不是一个面向连接的传输层协议，所以 CoAP 定义 4 种不同的报文类型：CON（Confirmable）、NON（Non-Confirmable）、ACK（Acknowledgement）和 RST（Reset）。本质来说，CoAP 采用了双层结构——消息层和请求 / 响应层。消息层处理端点之间的数据交换，并为 CON、NON、ACK 和 RST 报文类型提供重传机制，CoAP 通过增加消息层的方式弥补 UDP 传输的不可靠性。CoAP 的逻辑分层结构如图 5-4 所示。

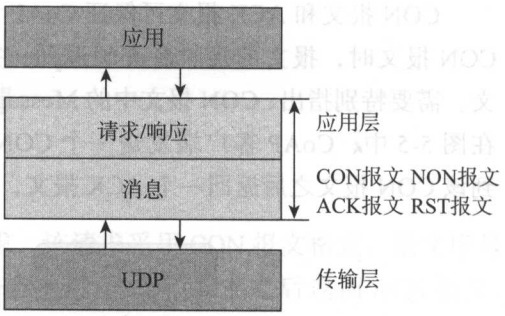


图 5-4　CoAP 协议逻辑结构

5.3.2　报文类型

CoAP 定义了四种不同的报文类型，通过这四种不同的报文类型至少可以组合成可靠传输和非可靠传输两种工作方式。

1. 4 种报文类型

- Confirmable Message（CON）：在请求 / 响应交互过程中，CON 报文需要被接收者确认。每一个 CON 报文必须要对应一个准确的 ACK 报文或 RST 报文，如果在规定的时间内客户端未接收到 ACK 报文或 RST 报文，那么客户端将会触发一次“重传”。
- Non-Confirmable Message（NON）：相比于 CON 报文，NON 报文的约束条件要宽松许多。单个 NON 报文不需要对应一个准确的 ACK 或 RST 报文。在一些传感器数据上传应用场景中 NON 报文较为常用，服务器对 NON 报文的处理也较为灵活。若服务器收到一个来自客户端的 NON 报文类型的 CoAP 请求，服务器可选择不返回响应；同时客户端也不会因为一定时间内没有收到来自服务器的 ACK 类型报文而触发重传机制。
- Acknowledgement Message（ACK）：ACK 报文用于确认 CON 报文。ACK 报文的报文序号（Message ID）需要和 CON 报文的报文序号保持严格一致，ACK 报文可以在一定程度上保证 CoAP 传输的可靠性。ACK 报文中可以不包括响应负载，可能为空。CoAP 请求 / 响应过程定义了携带模式和分离模式，ACK 报文负载为空的情

况一般出现在分离模式中。

- Reset Message（RST）：若服务器接收到一个 CON 报文，但由于报文中上下文缺失导致服务器无法处理该报文，那么服务器将会返回一个 RST 报文。RST 报文的报文序号需要与 CON 报文的报文序号保持严格一致，而且 RST 报文的负载一定为空。

2. 可靠传输

由于 CoAP 使用 UDP 作为传输层协议，UDP 并不是一种面向连接的传输层协议，所以对于一些要求可靠传输的场合，CoAP 需要另外增加某种机制保证传输的可靠性。

CON 报文和 ACK 报文可保证 CoAP 请求 / 响应交互过程的可靠性。当客户端发送一个 CON 报文时，报文的接收者必须返回一条 ACK 报文来确认其已经正确收到了该 CON 报文。需要特别指出，CON 报文中的 Message ID 必须与 ACK 报文中的 Message ID 完全一致。在图 5-5 中，CoAP 客户端发送一个 CON 报文，报文的 Message ID 为 0x7d34，接收者收到该 CON 报文之后返回一个 ACK 报文，ACK 报文的 Message ID 同为 0x7d34。

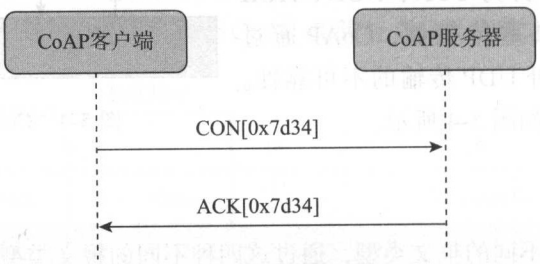


图 5-5　CoAP 可靠传输示例

若客户端在一定的时间内没有收到 ACK 报文，那么客户端将重新发送 CON 报文。CoAP 重传机制也可提高 CoAP 的可靠性，CoAP 重传机制通过两个主要的参数进行控制——超时时间和重传计数器，CoAP 重传机制将在 5.4 节详细分析。

3. 非可靠传输

在物联网应用领域并不是所有的发送者报文都需要被确认。NON 报文是一种非常轻量级的替换方案。如图 5-6 所示便是一个非可靠传输示例，CoAP 客户端发送一条 NON 报文，NON 报文的 Message ID 为 0x01a0，而服务器什么都没有返回。

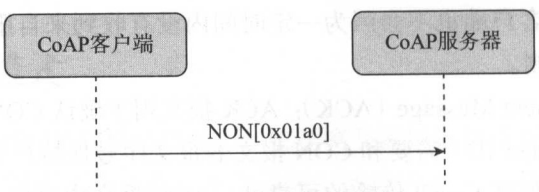


图 5-6　CoAP 非可靠传输示例

5.3.3　请求 / 响应模式

虽然 CoAP 参考了很多 HTTP 的优秀特性，但在具体实现过程中仍和 HTTP 存在区别。在 CoAP 中包含至少三种不同的请求 / 响应工作模式——携带模式、分离模式和非确认模式，其中携带模式和 HTTP 中的请求 / 响应模式最为相近。下面通过一个获取温度传感器检测结果的示例说明携带模式、分离模式和非确认模式的区别和联系。从 5.2 节中可获知，CoAP 首部中一定存在报文序号 Message ID 区域，该区域占 2 字节，而在 CoAP 首部中 Token 区域为可选择项，其内容和长度均由应用决定。在本小节的示例中，包含以下假定条件：

- 假定 Token 的长度为 1 字节。
- 假设温度传感器在服务器的资源 URI 为“temperature”。
- 通过 GET 方法可获取温度传感器检测结果。
- 检测结果采用文本格式表示。

1. 携带模式

在图 5-7 中，CoAP 客户端发送一次 GET 请求，该请求采用 CON 报文格式，报文序号 Message ID 为 0xbc90，Token 为 0x71；CoAP 服务器接收到 GET 请求之后返回 ACK 报文，ACK 报文的响应码为“2.05”，表示服务器正确处理请求并返回正确响应，ACK 报文的 Message ID 和 Token 与 GET 请求中的 Message ID 和 Token 完全相同。此时 ACK 报文中包含响应负载，响应负载为文本形式。

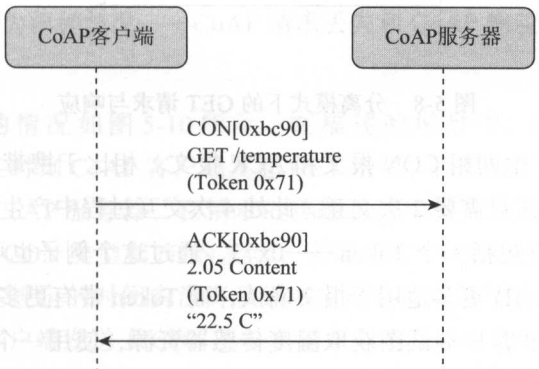


图 5-7　携带模式下的 GET 请求与响应

携带模式可简单理解为 ACK 报文中包含响应负载，例如此处的温度传感器检测结果——“22.5 C”（字符串形式描述）。携带模式是最常用的请求 / 响应工作模式，在这种工作模式下，Message ID 和 Token 的作用几乎相同。为了减少 CoAP 报文长度，更多的情况下只使用 Message ID 便可。

2. 分离模式

在图 5-8 中，CoAP 客户端发送一次 GET 请求，该请求同样采用 CON 报文格式，Message

ID 为 0x7a10，Token 为 0x73；CoAP 服务器接收到 GET 请求之后立刻返回 ACK 报文，但 ACK 报文中并没有包含任何响应负载。一段时间之后，服务器再向客户端发送一个 CON 报文，该报文的 Message ID 为 0x23bb，Token 为 0x73，该 CON 报文的响应码为“2.05”，表示服务器正确处理请求并返回正确响应。该 CON 报文中还包括响应负载，响应负载为文本形式的“22.5 C”。客户端收到 CON 报文之后，返回一个 ACK 报文进行确认。

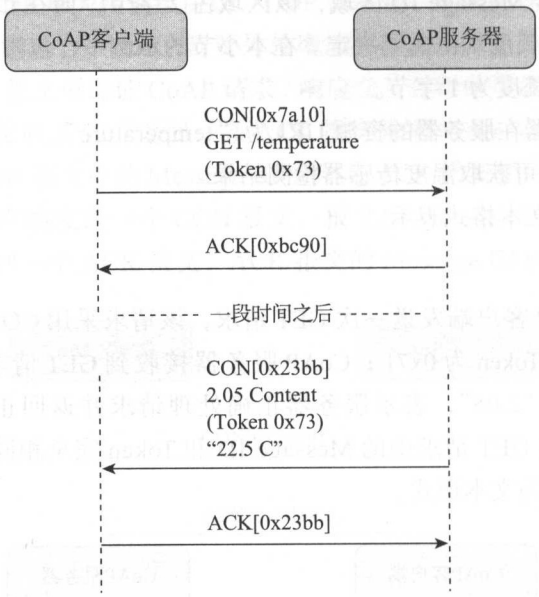


图 5-8　分离模式下的 GET 请求与响应

分离模式中总共产生两组 CON 报文和 ACK 报文。相比于携带模式，分离模式需要进行 4 次交互，而携带模式只需要 2 次交互。此处 4 次交互过程中产生了两个 Message ID——0x7a10 和 0x23bb，但仅包括一个 Token——0x73，通过这个例子也可以看出 Message ID 和 Token 的区别，Message ID 更多地用于报文确认，而 Token 带有更多的“应用”含义，可以理解为应用确认。CoAP 客户端试图获取温度传感器资源，使用一个 Token“标记”该应用动作，而 CoAP 服务器返回温度传感器检测结果时再把该 Token“标记”到响应内容中，这样 CoAP 客户端就可以通过该 Token 识别应用动作。

3. 非确认模式

非确认模式是 CoAP 中最为松散的请求 / 响应工作模式。在图 5-9 中，CoAP 客户端发送一次 GET 请求，但该报文并不是 CON 报文，而是一个并不需要确认的 NON 报文。CoAP 服务器接收到该 NON 报文之后，同样返回一个 NON 报文。非确认模式的其他部分与携带模式、分离模式非常相似。

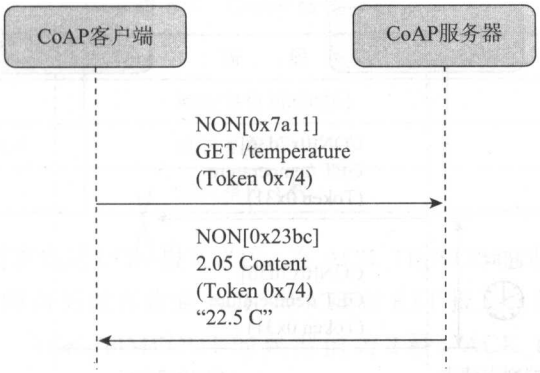


图 5-9　非确认模式

5.4　CoAP 重传机制

在网络数据传输过程中，无论是 CoAP 请求还是 CoAP 响应均可能产生丢失。CoAP 为了保证传输的可靠性还设计了重传机制。

5.4.1　CoAP 重传情况分析

分析 CoAP 重传机制之前我们先来分析网络数据传输过程中丢失的情况。对于 CoAP 来说网络数据丢失可分为两种情况——CoAP 请求丢失和 CoAP 响应丢失。

1. CoAP 请求丢失

CoAP 请求丢失的情况如图 5-10 所示。在描述的场景中，CoAP 客户端试图获取 temperature 资源。在某时刻 CoAP 客户端发起 GET 请求，但是该 CoAP 请求并没有如客户端预期那样准确到达 CoAP 服务器，在请求的传输过程中报文意外丢失。通过 5.3 节可以获知，CoAP 请求分为 CON 类型和 NON 类型，若请求为 CON 类型那么客户端必须等待来自服务器的响应，若在规定的时间内没有收到响应那么客户端便会认为请求失败。在如图 5-10 所示的场景中，客户端发送了 CON 类型的请求但是在一定的时间内没有收到服务器的响应，那么客户端将会重新发送 CoAP 请求，而且两次 CoAP 请求的首部和负载完全相同。

2. CoAP 响应丢失

除了 CoAP 请求丢失的情况之外，CoAP 客户端还需要处理响应丢失的情况。如图 5-11 所示便是典型的 CoAP 响应丢失场景。CoAP 客户端试图获取服务器中的 temperature 资源，它向服务器发送 CON 类型的 GET 请求，服务器正常返回了响应但该响应没有正确到达客户端。对于客户端来说，由于在规定时间内没有收到响应，那么前一次 CoAP 请求将会判定为失败。为了正确获取 temperature 资源，客户端不得不再次发送 CoAP 请求，而且两次的 CoAP 请求完全一致。此时 CoAP 服务器只能“不厌其烦”地返回相同的内容。

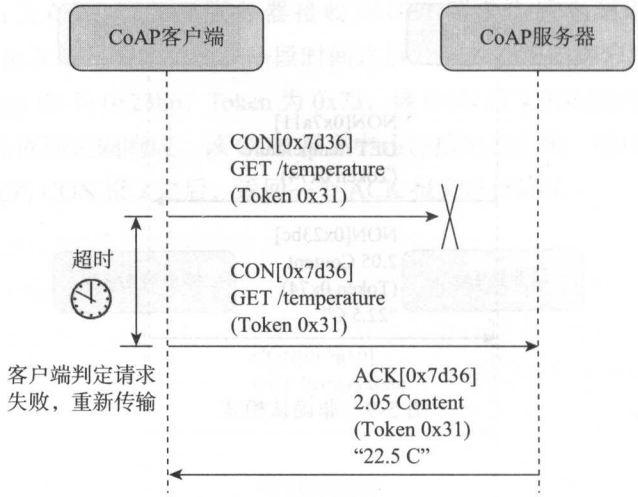


图 5-10 请求丢失

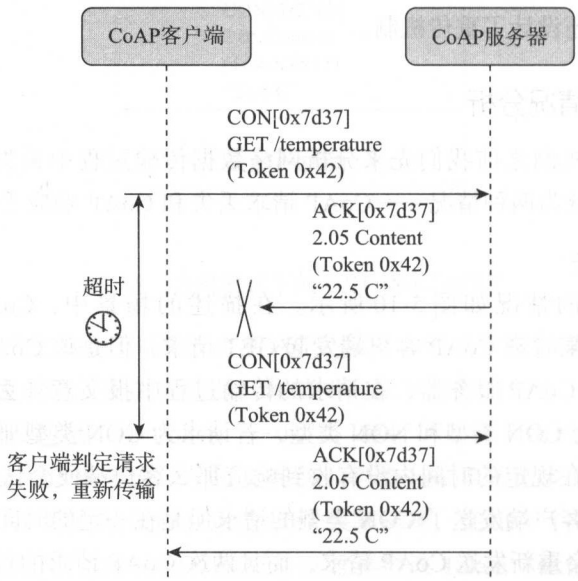


图 5-11 响应丢失

5.4.2 传输参数说明

通过前面的分析可以获知 CoAP 请求和响应均有丢失的风险，为了解决该问题，CoAP 客户端需要重新发送 CoAP 请求。CoAP 报文重传一般由 ACK_TIMEOUT、ACK_RANDOM_FACTOR 和 MAX_RETRANSMIT 三个参数控制。CoAP 传输参数见表 5-1。

表 5-1　CoAP 传输参数

参数名称	说　明	典　型　值
ACK_TIMEOUT	响应等待超时时间	2 秒
ACK_RANDOM_FACTOR	随机系数	1.5
MAX_RETRANSMIT	最大重传次数	4

如果 CoAP 客户端首次发送 CON 报文之后，在 ACK_TIMEOUT 到 ACK_TIMEOUT*ACK_RANDOM_FACTOR 时间内仍没有收到 ACK 报文或者 RST 报文，那么 CoAP 客户端将重新发送一次 CON 报文。ACK_TIMEOUT 的典型值为 2 秒。ACK_RANDOM_FACTOR 是一个不能小于 1.0 的随机数，该参数的典型值为 1.5。MAX_RETRANSMIT 定义最大重传次数，该参数的典型值为 4。在极端情况下，CoAP 客户端将会执行 4 次重传，加上第一次 CON 报文，总共将产生 5 次 CoAP 请求。

CoAP 重传机制由超时时间和重传计数器两个参数控制。对于一个 CON 报文来说，初始的超时时间为 ACK_TIMEOUT 到 ACK_TIMEOUT*ACK_RANDOM_FACTOR 之间的随机数，如果 ACK_TIMEOUT 采用典型值 2 秒，ACK_RANDOM_FACTOR 采用典型值 1.5，那么初始超时为 2～3 秒，如 2.45 秒、2.95 秒都是合理的初始超时时间。重传计数器从 0 开始递增，一旦在规定的时间内没有收到 ACK 报文或 RST 报文，那么 CON 报文将会被重新发送，重传计数器自动增加。下一次重传超时时间为上一次超时时间的两倍，如初始重传时间为 2.45 秒，那么第一次重传的超时时间为 4.90 秒。在 CoAP 协议中重传超时时间将越来越大。

我们通过一个更加具体的例子说明 CoAP 重传机制，假设 ACK_TIMEOUT 取值为 2，ACK_RANDOM_FACTOR 取值为 1.3，MAX_RETRANSMIT 取值为 4，那么超时等待时间见表 5-2。

表 5-2　超时等待时间计算示例

第 *n* 次等待	超时等待时间 / 秒
第 1 次等待	2×1.3=2.6
第 2 次等待	2×2.6=5.2
第 3 次等待	2×5.2=10.4
第 4 次等待	2×10.4=20.8
第 5 次等待	2×20.8=41.6

5.4.3　最大传输耗时（MAX_TRANSMIT_SPAN）

假设 ACK_TIMEOUT 采用典型值 2 秒，ACK_RANDOM_FACTOR 采用典型值 1.5，MAX_RETRANSMIT 同样采用典型值 4，超时重传时间的初始值采用随机结果的上限也就

是 ACK_TIMEOUT*MAX_RETRANSMIT。最糟糕的情况下，CoAP 客户端传输一次正常的 CON 报文并且重传 4 次 CON 报文，在该过程中相邻 CON 报文的时间间隔分别为 ACK_TIMEOUT*1.5、2*ACK_TIMEOUT*1.5、4*ACK_TIMEOUT*1.5、8*ACK_TIMEOUT*1.5，总计时间为：

MAX_TRANSMIT_SPAN＝ACK_TIMEOUT*1.5*(1＋2＋4＋8)＝45（秒）

图 5-12 可以很好地说明该重传过程。CoAP 中最大传输耗时的典型值为 45 秒，在实际使用过程中，由于 ACK_RANDOM_FACTOR 为一个 1～1.5 之间的随机值，所以最大传输耗时往往小于 45 秒。

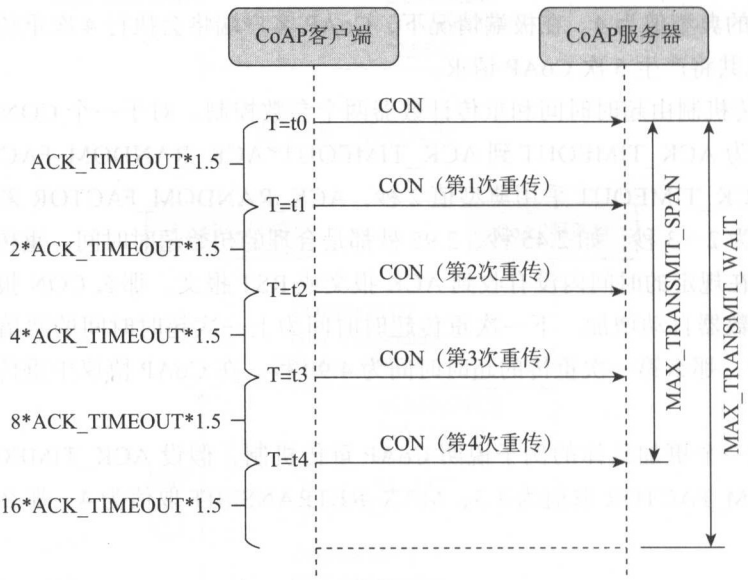


图 5-12　CoAP 重传机制说明

5.4.4　最大等待时间（MAX_TRANSMIT_WAIT）

当 CoAP 客户端执行最后一次重传之后并不意味着整个传输过程已经结束，CoAP 客户端还需要等待最后一次超时时间。最后一次超时时间为 16*ACK_TIMEOUT*1.5，那么从 CoAP 客户端发送第一次 CON 报文到最后结束等待总共消耗的时间为：

MAX_TRANSMIT_WAIT＝ACK_TIMEOUT*1.5*(1＋2＋4＋8＋16)＝93（秒）

图 5-12 可以说明最大传输耗时（MAX_TRANSMIT_SPAN）与最大等待时间（MAX_TRANSMIT_WAIT）之间的联系与区别。

5.5 CoAP方法

CoAP共有四种不同的请求方法：GET方法、POST方法、PUT方法和DELETE方法。每一个不能被服务器识别或未被服务器支持的方法通常会导致一个4.05响应码。

5.5.1 GET

GET方法用于查询资源，该资源通过请求中的URI进行识别。由于单个资源服务器可能使用不同的媒体类型展示某资源，所以GET请求中可包括一个或多个Accept选项用于指示客户端期望获得的媒体类型。若GET请求被正确执行，2.05（Content）或2.03（Valid）响应码将会出现在CoAP响应报文中。无论是在HTTP应用中还是在CoAP应用中，GET总是最常用的方法。

5.5.2 POST

POST方法要求CoAP请求中的资源描述内容被CoAP服务器处理。如果POST请求被正确执行，那么在服务器上将创建一个新资源。一旦在服务器上创建一个新资源，服务器返回的响应中将包括一个2.01（Create）响应码并且响应负载中包括一个新建资源的URI，该URI可使用一个或多个Location-Path和Location-Query定义。如果POST请求被成功执行但未在服务器上创建新资源，那么响应消息中应包含一个2.04（Changed）响应码；如果POST请求被成功执行但导致服务器上的目标资源被删除，那么响应消息中应包含一个2.02（Deleted）响应码。

5.5.3 PUT

PUT方法要求服务器根据CoAP请求中的URI和CoAP请求负载中的资源描述信息更新服务器内的指定资源。如果资源已经存在，那么服务器将会更新指定资源，同时返回一个包含2.04（Changed）响应码的CoAP响应；如果资源不存在，那么服务器将会根据请求中的URI创建一个新的资源，同时返回一个包含2.01（Created）响应码的CoAP响应。如果该资源既不能被创建也不能被更新，那么服务器将返回一个合适的错误响应码。

5.5.4 DELETE

DELETE方法要求服务器根据CoAP请求中的URI删除服务器中的指定资源。如果资源删除成功，那么服务器应返回一个包含2.02（Deleted）响应码的CoAP响应。

5.6 CoAP响应码

与HTTP类似，CoAP也包含三种不同类型的响应码——2.xx Success（成功）、4.xx

Client Error（客户端错误）和 5.xx Server Error（服务器错误）。CoAP 响应码和 HTTP 状态码存在很强的对应关系。

5.6.1 正确响应

若服务器正确执行 CoAP 请求，将会返回一个表示执行正确的响应码，这些响应码包括 2.01 Created、2.02 Deleted、2.03 Valid、2.04 Changed 和 2.05 Content。

1. 2.01 Created

类似于 HTTP 201 Created，该响应码只能应用于 POST 或 PUT 响应中，表示服务器创建了一个新的资源。2.01 Created 和 2.04 Changed 响应码均可应用于 POST 或 PUT 响应中。但 2.01 Created 表示创建资源，资源的状态从无到有；而 2.04 Changed 表示更新资源，资源的状态从旧到新。

2. 2.02 Deleted

类似于 HTTP 204 No Content，该响应码只能应用于 DELETE 响应中，表示服务器成功删除一个资源。

3. 2.03 Valid

类似于 HTTP 304 Not Modified，2.03 Valid 和 2.05 Content 常用于 GET 响应中，但两者存在一些区别，包含 2.03 Valid 响应码的 CoAP 响应中负载内容一般为空，而包含 2.05 Content 响应码的 CoAP 响应中一般包含具体负载。

4. 2.04 Changed

类似于 HTTP 204 No Content，该响应码只能应用于 POST 和 PUT 响应中，表示服务器更新了某个资源。

5. 2.05 Content

类似于 HTTP 200 OK，该响应码只能应用于 GET 响应中，这可能是客户端最愿意看到的响应码。

5.6.2 客户端错误

若 CoAP 服务器发现客户端请求错误，将返回一个包含客户端错误的响应码。这些响应码包括 4.00 Bad Request、4.01 Unauthorized、4.02 Bad option、4.03 Forbidden、4.04 Not Found、4.05 Method Not Allowed、4.06 Not Acceptable 和 4.12 Precondition Failed。

1. 4.00 Bad Request

类似于 HTTP 400 Bad Request，4.00 Bad Request 表示通用客户端状态指示，当使用其他 4.XX 无法描述客户端错误时可直接使用 4.00 Bad Request。

2. 4.01 Unauthorized

该响应码表示客户端未获取权限去执行相关操作。客户端在没有提供适当的身份凭证的情况下向服务器中受保护的资源发送请求时，若服务器不想承认该资源存在于服务器，可返回 4.04 Not Found 代替 4.01 Unauthorized。

3. 4.02 Bad option

该响应码表示请求中包含一个或多个未能识别的选项。

4. 4.03 Forbidden

类型于 HTTP 403 Forbidden，该响应码表示客户端请求格式正确，但是服务器并不愿意执行该请求。

5. 4.04 Not Found

类似于 HTTP 404 Not Found，4.04 Not Found 表示服务器无法寻找到地址资源。

6. 4.05 Method Not Allowed

类似于 HTTP 405 Method NOT Allowed，4.05 Method Not Allowed 表示客户端使用一个未经服务器定义的 CoAP 方法访问某资源。例如该资源仅支持 POST 方法，但是客户端使用 PUT 方法修改资源，此时 CoAP 服务器将返回 4.05 Method Not Allowed。

7. 4.06 Not Acceptable

类似于 HTTP 406 Not Acceptable，即客户端请求中带有 Accept 选项，而服务器无法根据 Accept 选项返回指定内容。例如服务器内的某资源仅支持 JSON 格式或文本格式，但 CoAP 客户端却希望获得 XML 格式负载，由于 CoAP 服务器无法满足 CoAP 客户端的期望，只能返回 4.06 Not Acceptable。

8. 4.12 Precondition Failed

类似于 HTTP 412 Precondition Failed，客户端在请求中定义了一个或多个先决条件，例如在请求中增加 If-Match 选项，而服务器只有在满足特定条件下才可以处理该请求，若特定条件无法满足，服务器将返回 4.12 Precondition Failed。

5.6.3 服务器错误

若 CoAP 服务器由于自身问题无法正确执行 CoAP 请求，那么将会返回一个服务器错误响应码，这些响应码包括 5.00 Internal Server Error、5.01 Not Implemented 和 5.03 Service Unavailable。

1. 5.00 Internal Server Error

类似于 HTTP 500 Interal Server Error，5.00 Internal Server Error 是一个通用的服务器错误响应，若使用其他 5.XX 响应码无法正确描述错误可直接使用 5.00 Internal Server Error。

2. 5.01 Not Implemented

类似于 HTTP 501 Not Implemented，5.01 Not Implemented 表示 CoAP 服务器不支持请求中规定的某些特性。

3. 5.03 Service Unavailable

类似于 HTTP 503 Service Unavailable，5.03 Service Unavailable 意味着虽然 CoAP 服务器正常启用，但是某些应用并没有正常工作。例如，众多的 CoAP 客户端向服务器发送 CoAP 请求，CoAP 服务器无法及时处理所有的请求，在这种情况下 CoAP 服务器可返回 5.03 Service Unavailable 响应码。

5.7 CoAP 选项

CoAP 定义了一系列选项用以规范 CoAP 报文的格式。一个 CoAP 报文中可以包括多个 CoAP 选项。CoAP 选项是 CoAP 中最难理解的部分。

5.7.1 选项格式

CoAP 选项实例由选项偏移量（Option Delta）、选项长度（Option Length）和选项值（Option Value）组成。CoAP 选项的具体格式如图 5-13 所示。CoAP 中不能直接确定选项编号，选项编号必须由上一个选项编号和本次选项偏移量计算得到。

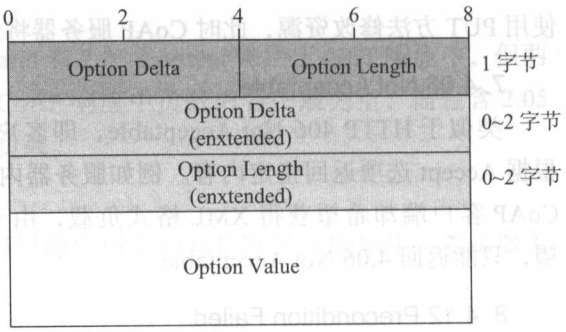


图 5-13 选项格式

1. 选项偏移量

4 位无符号整数。其中 0～12 用于指示选项的偏移量，13、14 和 15 具有特殊含义。

- 13：Option Delta extended 区域定义一个 8 位无符号整数，此时的选项偏移量应为该 8 位无符号整数 +13。
- 14：Option Delta extended 区域定义一个 16 位无符号整数，此时的选项偏移量应为该 16 位无符号整数 +269。
- 15：保留为将来使用。

选项偏移量描述了当前选项编号与之前选项编号之间的差值，换句话说选项编号可以由当前选项编号与之前的选项编号计算得到。例如，上次选项编号为 11（Uri-Path），当前选项偏移量为 4，那么当前的选项编号为 15（Uri-Query）。

2. 选项长度

4 位无符号整数。0～12 用于指定选项长度，13、14 和 15 具有特殊含义。

- 13：Option Length extended 区域定义一个 8 位无符号整数，此时的选项长度应为该 8 位无符号整数 +13。
- 14：Option Length extended 区域定义一个 16 位无符号整数，此时的选项长度应为该 16 位无符号整数 +269。
- 15：保留为将来使用。

3. 选项值

在 CoAP 中选项值包含四种数据类型——Empty、Opaque、Uint 和 String。

- Empty：选项值长度为 0。
- Opaque：选项值长度不确定。
- Uint：选项值长度为非负整数，该值采用网络字节顺序即大端格式定义。
- String：UTF-8 编码字符串格式。

4. 选项定义

CoAP 选项定义见表 5-3。

表 5-3　CoAP 选项定义

选 项 值	选 项 名 称	数 据 类 型	长度定义 / 字节
1	If-Match	Opaque	0～8
2	Uri-Host	String	1～255
4	E-Tag	Opaque	1～8
5	If-None-Match	Empty	0
7	Uri-Port	Uint	0～2
8	Location-Path	String	0～255
11	Uri-Path	String	0～255
12	Content-Format	Uint	0～4
14	Max-Age	Uint	0～2
15	Uri-Query	String	0～255
16	Accept	Uint	0～2
17	Location-Query	String	0～255

5.7.2　URI 相关选项

在 CoAP 选项中共有 4 个选项与 URI 直接相关——Uri-Host、Uri-Port、Uri-Path 和 Uri-Query。这些参数都可以用于定位服务器资源，CoAP 的资源定位规则也符合 URL 语法，例如：

```
coap://wsnccoap.org:5683/devices/1234CDEF?limit=10&offset=20
```

- Uri-Host 用于定义服务器名称，此 CoAP URI 示例中 Uri-Host 等于“wsnccoap.org”。

- Uri-Port 用于定义服务器 CoAP 服务端口号，此 CoAP URI 示例中 Uri-Port 的值为 5683。
- Uri-Path 用于定义资源在服务器中的相对或绝对位置，在此 CoAP URI 示例中 Uri-PATH 等于“devices/1234CDEF”。此时“devices/1234CDEF”被分成了两个 Uri-Path：一个为“devices”，另一个为“1234CDEF”。与其他 CoAP 选项实例一样，CoAP 选项编号和选项长度均被完整定义，所以 CoAP 选项中并没有“/”这样的分隔符，也就是说“devices”和“1234CDEF”之间的“/”绝不会出现在 CoAP 选项内。
- Uri-Query 用于定义资源的查询参数，在此 CoAP URI 示例中 Uri-Query 等同于“?limit=10&offset=20”，与 Uri-Path 中的情况相似，像“?”和“&”这样的分隔符并不会出现在 CoAP 选项内。此处也有两个 Uri-Query：一个为“limit=10”，另一个为“offset=20”。

5.7.3 Content-Format 选项

Content-Format 选项用于指示 CoAP 选项中的负载媒体类型，CoAP 负载媒体类型采用整数编号的方式定义，该编号采用无符号整数表示，不同的媒体类型采用不同的编号，如二进制负载的媒体类型编号为 42，JSON 负载的媒体类型编号为 50。Content-Format 和 HTTP 中的 Content-Type 非常相似，在 HTTP 中若表示负载为 JSON 格式，需要在 HTTP 首选项中加入“Content-Type:Application/json\r\n”，在传输过程中该部分至少需要占用 31 字节，而在 CoAP 中却只占用 3 字节。CoAP 媒体类型的相关内容请参考 5.8 节。

5.7.4 Accept 选项

Accept 选项用于表示 CoAP 客户端期望接收到的媒体类型格式，Accept 选项的负载类型定义和 Content-Format 选项的负载类型定义完全相同。如果 CoAP 请求中没有包含 Accept 选项，那么 CoAP 服务器将返回默认媒体类型；若 CoAP 选项中包含 Accept 选项，CoAP 服务器根据客户端期望的媒体类型返回响应负载，如果服务器不能返回指定格式的响应，那么可在 CoAP 响应中加入 4.06（Not Acceptable）响应码。

通常情况下资源可以采用不同的媒体类型描述，例如一个温度传感器资源可以使用文本类型描述，也可以采用 JSON 格式描述，更可以使用 2 字节无符号整数这样的二进制方式描述。若采用二进制方式描述响应负载，需要提前定义大小端格式，在网络传输过程中一般采用大端格式。图 5-14 展示了 Accept 选项和 Content-Format 选项如何配合使用。

5.7.5 Etag 选项

与 HTTP 相似，CoAP 中也包含 Etag 选项，而且 CoAP 中的 Etag 选项与 HTTP 中 Etag 首部字段的使用方法非常相似。Etag 可以理解为资源的实体标记，它用来表示资源的新鲜程度，当资源发生改变时 Etag 的值也需要更新。

无论是在 CoAP 中还是在 HTTP 中，Etag 并没有统一的计算规则，但是无论如何 Etag

的具体值都是由服务器分配。Etag可以有很多方法生成，如版本号、校验和、单向散列值和时间参数等。在实际应用中，单向散列值往往是最受欢迎的计算Etag的方法。

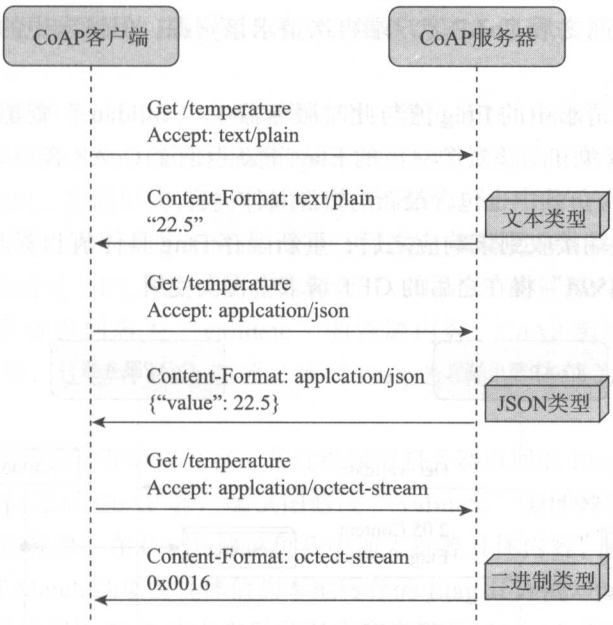


图5-14 Accept选项和Content-Format选项

Etag既可以出现在CoAP请求中，也可出现在CoAP响应中，而且在CoAP中，Etag的具体值还经常与If-Match选项配合使用。虽然CoAP中的Etag和HTTP中的Etag首部字段的用法相似，但是CoAP中Etag的具体值一般为二进制形式，而HTTP的Etag首部选项一般采用字符串形式。下面通过一个例子说明Etag在CoAP中的使用方法，具体过程如图5-15使用。

1）CoAP客户端通过GET方法试图获取名为“validate”的资源内容。

2）CoAP服务器将返回名为“validate”的资源体内容，CoAP响应中除了包含“2.05 Content”响应码之外，还包含Etag选项，此时Etag选项的具体值为二进制形式表示的“0x9868”。

3）CoAP客户端将会缓存响应内容，并且将保留服务器返回的Etag。

4）经过一段时间之后CoAP客户端再次请求“validate”资源，由于CoAP客户端已经缓存该资源的Etag具体值，所以在本次请求中CoAP请求首部中加入了Etag选项。

5）CoAP服务器收到请求之后取出请求首部中的Etag值，CoAP服务器发现请求中的Etag值与服务器中“validate”的Etag值完全相同，说明资源并没有被更新。此时CoAP服务器返回的响应码为“2.03 Valid”，但CoAP响应中并没有响应内容。

6）CoAP客户端接收到服务器的响应，发现响应码为“2.03 Valid”，说明客户端请求

的资源并没有改变，直接使用之前的缓存即可。

7）某时刻 CoAP 服务器中的“validate”资源被改变，服务器重新计算该资源的 Etag，Etag 值被更新为二进制形式的“0x559d”。

8）经过一段时间之后 CoAP 客户端再次请求该资源，但请求中的 Etag 具体值依然为原缓存值“0x9868”。

9）服务器发现请求中的 Etag 值与此时服务器中“validate”资源的最新 Etag 值不相符，于是把最新的资源和与该资源对应的 Etag 重新返回至 CoAP 客户端，那么响应码变为“2.05 Content”，响应负载中也包含最新的资源具体内容。

10）CoAP 客户端接收到该响应之后，重新缓存 Etag 具体值和资源具体内容。更新之后 Etag 具体值“0x559d”将在之后的 GET 请求中再次使用。

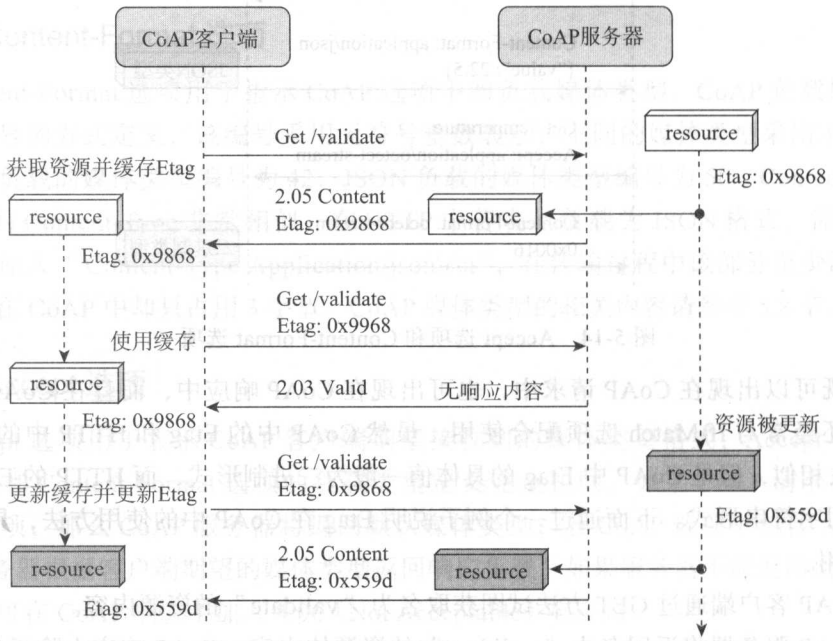


图 5-15 Etag 使用示例

在 CoAP 中使用 Etag 选项需要明确以下几点。

- Etag 选项一般配合 GET 方法使用。
- Etag 选项可以节省响应内容，让 CoAP 客户端充分利用本地缓存。
- Etag 选项既可以出现在 CoAP 请求中，也可以出现在 CoAP 响应中。
- Etag 选项总是由服务器生成，客户端必须无条件缓存 Etag。

5.7.6 If-Match 选项

如果请求中包含 If-Match 选项或 If-None-Match 选项，称这类 CoAP 请求为 CoAP 条

件请求，If-Match 选项一般用于更新服务器资源，其具体值一般采用 Etag 选项的具体值。

If-Match 选项的使用方法也非常简单，如果服务器收到的 If-Match 选项值与被更新资源的 Etag 值相同，则认为该条件请求有效，响应码为“2.04 change”；如果服务器收到的 If-Match 选项值与被更新资源的 Etag 值不相同，则认为该条件请求无效，响应码为“4.12 Precondition Failed”。

HTTP 中也有相似的条件请求，这些条件请求一般包括 If-Match、If-None-Match 和 If-Modify-Since 等请求首部字段。CoAP 中条件请求的使用方法与 HTTP 相似，但 CoAP 中的相关选项总是优先使用二进制形式，这点与 HTTP 中各种首部字段存在明显差异。

下面通过一个具体的例子说明 If-Match 的使用方法，具体过程如图 5-16 所示。

1）CoAP 客户端通过 GET 方法试图获取名为“validate”的资源内容。

2）CoAP 服务器将返回名为“validate”的资源内容，CoAP 响应中除了包含“2.05 Content”响应码之外，还包含 Etag 选项，此时 Etag 选项的具体值为二进制形式表示的“0x559d”。

3）CoAP 客户端将会缓存响应内容，并且将保留服务器返回的 Etag。

4）经过一段时间之后 CoAP 客户端试图修改“validate”资源内容，此时 CoAP 客户端通过 PUT 方法更新资源，在 CoAP 请求负载中加入资源具体内容，在 CoAP 选项中加入 If-Match 选项，而 If-Match 选项的具体值为本地缓存的 Etag 具体值“0x559d”。

5）CoAP 服务器收到该 PUT 请求之后发现请求首部中 If-Match 选项的具体值与本地资源的 Etag 值完全相同，所以认为该 PUT 请求合法。CoAP 服务器将使用客户端请求负载中的内容更新资源，并重新计算该资源的 Etag 值。CoAP 响应中响应码为“2.04 Changed”，CoAP 响应中还包括更新之后的 Etag 具体值“0x6cfa”。

6）CoAP 客户端收到该响应，缓存了响应中的 Etag 具体值“0x6cfa”以备下次使用。

7）在某一时刻其他 CoAP 客户端修改了“validate”资源，服务器中“validate”资源的 Etag 具体值被重新计算为“0x2f09”。

8）经过一段时间之后 CoAP 客户端试图再次更新“validate”资源，在 CoAP 请求中使用 If-Match 选项，该选项的具体值为之前缓存的 Etag 具体值“0x6cfa”。

9）CoAP 服务器接收到该请求之后发现 If-Match 选项的具体值与资源 Etag 的具体值并不匹配，此时 CoAP 服务器认为该客户端并不能更新该资源，所以在响应中响应码变更为“4.12 Precondition Failed”。

10）CoAP 客户端收到该响应后知晓本次 PUT 请求失败，如果需要正确更新“validate”资源，需要先发起 GET 请求获取最新资源内容，重新修改之后再发起 PUT 请求。

在 CoAP 中使用 If-Match 选项需要明确以下几点。

- If-Match 选项一般和 Etag 具体值配合使用。
- 在 CoAP 请求中，If-Match 选项往往配合 PUT 请求或 POST 请求使用，而 Etag 选项配合 GET 请求使用。

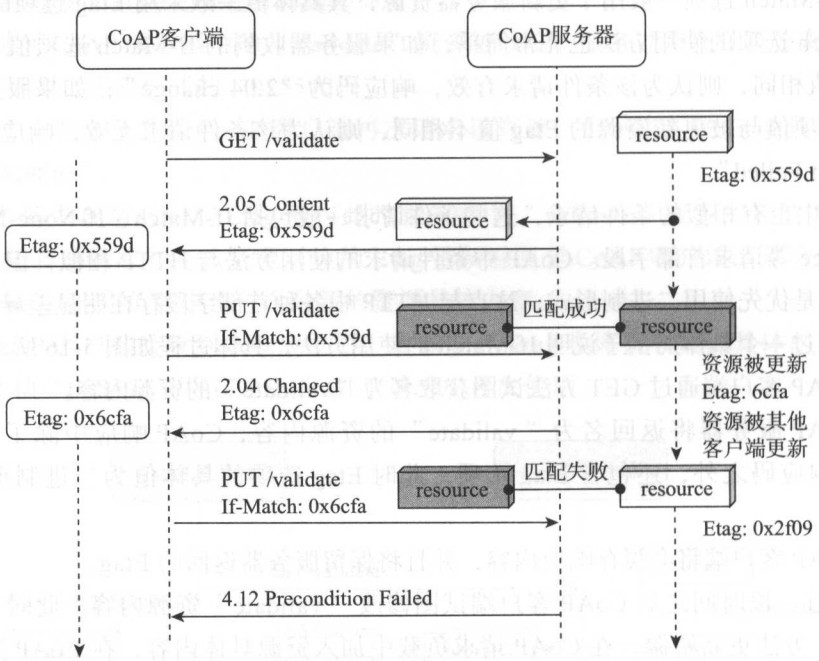


图 5-16　If-Match 示例

- If-Match 选项一般用于更新已经存在的资源，所以往往需要通过 GET 请求获取资源的 Etag 具体值。

注意　此处的请求失败与 CoAP 重传中提到的请求失败的含义完全不同，在 CoAP 重传中请求失败是由于传输过程导致的，而此处的请求失败是由于应用处理过程导致的，两者的含义完全不同。

5.7.7　If-None-Match 选项

If-None-Match 选项和 If-Match 选项的使用方法相似，If-None-Match 选项一般用于在服务器上创建一个新资源，而 If-Match 选项一般用于修改已经在服务器中存在的资源。下面依然通过一个示例说明如何使用 If-None-Match 选项，具体过程如图 5-17 所示。

1）CoAP 客户端 A 试图在服务器中创建“create1”资源，在 CoAP 请求中增加了 If-None-Match 选项。CoAP 客户端 A 在服务器上成功创建了该资源，服务器返回 CoAP 响应的响应码为“2.01 Created”。

2）CoAP 客户端 B 也试图在服务器中创建“create1”资源，在 CoAP 请求中也增加了 If-None-Match 选项，但是此时服务器已经创建了“create1”资源，所以无法执行该 CoAP 请求，服务器返回 CoAP 响应的响应码为“4.12 Precondition Failed”。

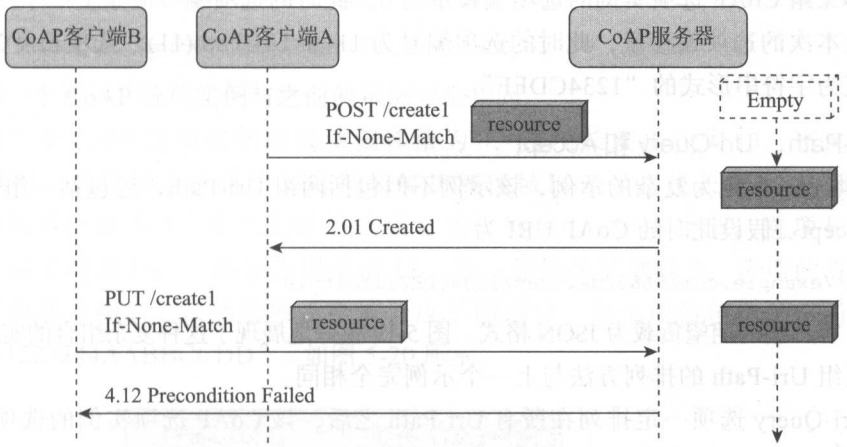


图 5-17 If-None-Match 使用示例

5.7.8 选项示例

CoAP 选项部分非常复杂，为了更好地说明 CoAP 选项的用法，下面列举实际情况中经常出现的几种 CoAP 选项组合方式。

1. Uri-Path

第一个示例 CoAP 选项中包含两组 Uri-Path，假设此时 CoAP URI 为

```
coap://example.com:5683/devices/1234CDEF
```

通过前文的描述获知，在 CoAP 请求中一般可以省略 Uri-Host 和 Uri-Port，所以本例中仅包括两个 Uri-Path：“devices”和“1234CDEF”，图 5-18 展示了这两个 Uri-Path 在 CoAP 首部中的具体排列方式。

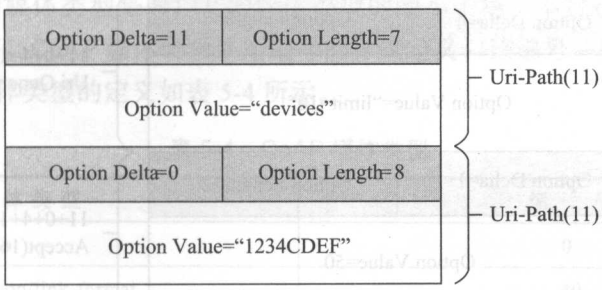


图 5-18 Uri-Path 示例

- 第一组 CoAP 选项实例的选项偏移量为 11，由于该部分是 CoAP 首部中的第一个选项，那么此时的选项偏移量就是选项编号，此时的选项编号为 Uri-Path(11)。选项长度为 7，选项值为字符串形式的“devices”。

- 第二组 CoAP 选项实例的选项偏移量为 0，此时的选项编号应是上一个选项编号加上本次的选项偏移量，此时的选项编号为 11+0=Uri-Path(11)。选项长度为 8，选项值为字符串形式的“1234CDEF”。

2. Uri-Path、Uri-Query 和 Accept

下面构造一个更为复杂的示例，该示例不但包括两组 Uri-Path，还包括一组 Uri-Query 和一组 Accept。假设此时的 CoAP URI 为

```
coap://example.com:5683/devices/1234CDEF?limit=10
```

CoAP 客户端的期望负载为 JSON 格式。图 5-19 很好地展现了这种复杂组合的实现细节。

- 两组 Uri-Path 的排列方法与上一个示例完全相同。
- Uri-Query 选项一定排列在所有 Uri-Path 之后，该 CoAP 选项实例的选项偏移量为 4，此时的选项编号为之前所有选项偏移量之和 11+0+4=Uri-Query(15)。选项长度为 8，选项值为字符串形式的“limit=10”。
- Accept 选项一定排列在 Uri-Path 之后，该选项的选项偏移量为 1，该 CoAP 的选项编号为之前所有选项偏移量之和 11+0+4+1=Accept(16)。选项长度为 2，选项值为 2 字节无符号整数 50。

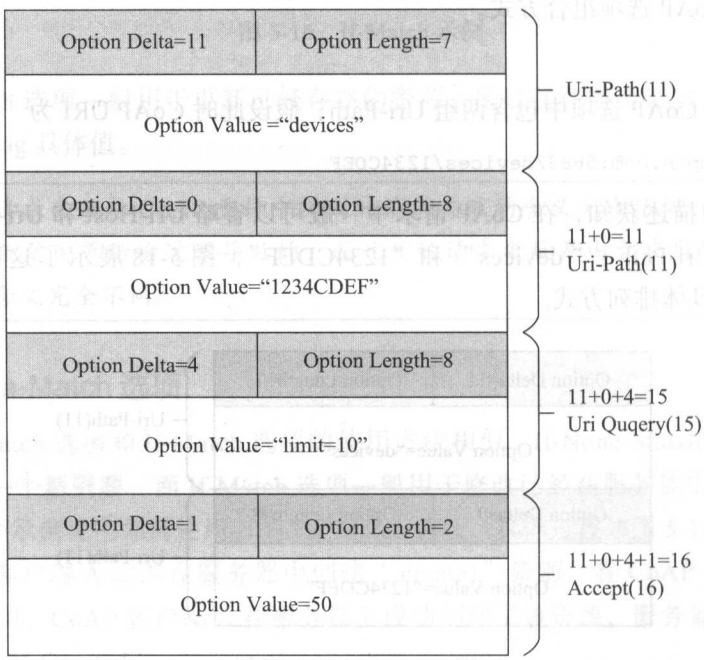


图 5-19　Uri-Path、Uri-Query 和 Accept 示例

3. 选项长度扩展

有时 CoAP URI 较长，单个 Uri-Path 的长度可能超过 12 字节，在下面的 URI 中，第二

个 Uri-Path“11223344AABBCCDD”长度为 16 字节，此时便需要使用选项长度扩展部分：

```
coap://example.com:5683/devices/11223344AABBCCDD
```

- 第一个 CoAP 选项实例与之前的示例完全相同。
- 第二个 CoAP 选项实例的选项偏移量为 0，选项长度指示为 13。当选项长度为 13 时，选项值之前将增加选项长度扩展区域以表示超过长度 13 的部分，该扩展区域的值为 3，那么此时的选项真实长度为 13+3=16 字节。选项长度指示被分成了两部分：一部分为固定值 13，另一部分为扩展值 3，选项的真实长度需将两部分相加。虽然选项长度被分成了两部分，但选项值依然为字符串形式的“11223344AABBCCDD”。如图 5-20 所示。

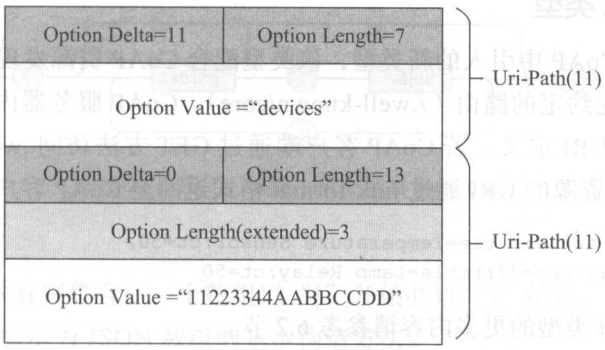


图 5-20　选项长度扩展

5.8　CoAP 媒体类型

CoAP 支持多种媒体类型，但 CoAP 支持的媒体类型数量远小于 HTTP 支持的媒体类型，可以说 CoAP 的媒体类型是 HTTP 的媒体类型的微小子集，不过在物联网领域这些媒体类型已经完全足够。CoAP 媒体类型采用编号的方式定义，该编号一般采用 2 字节无符号整数定义。CoAP 媒体类型的定义如表 5-4 所示。

表 5-4　CoAP 媒体类型

媒体类型	编　号
text/plain	0
application/link-format	40
application/xml	41
application/octet-stream	42
application/exi	47

（续）

媒体类型	编　号
application/json	50
application/cbor	60

在物联网实际应用中，文本类型 text/plain、二进制类型 application/octet-stream、JSON 类型 application/json 应用最为广泛，二进制 JSON 类型 application/cbor 也逐渐投入使用。application/link-format 是一种专属于 CoAP 的媒体类型，该媒体类型一般在 CoAP 资源发现中使用。

5.8.1 link-format 类型

link-format 是 CoAP 中引入的新类型，该类型配合 CoAP 资源发现使用。CoAP 服务器中一般包括一个事先约定的路由（/.well-known/core）。CoAP 服务器内一般包括较多资源，这些资源由不同的 URI 定义。若 CoAP 客户端通过 GET 方法访问 .well-known/core 路由，CoAP 服务器将相关资源的 URI 通过 link-format 格式返回至 CoAP 客户端，例如：

```
<sensors/temp>;sz=64;title=Temperature Sensor;ct=50,
<actuators/relay>;sz=32;title=Lamp Relay;ct=50
```

关于 link-format 类型的更多内容请参考 6.2 节。

5.8.2 文本与二进制类型

文本类型和二进制类型是较为简单的媒体类型。CoAP 中的默认媒体类型为文本类型。若负载为文本类型，无论在 CoAP 请求或 CoAP 响应中均不需要指定 Content-Format。虽然文本类型使用非常简单，但是也存在一定的局限性。例如使用文本类型表示三个传感器检测结果，可能采用这样的方式定义负载内容：

```
12.3,80.5,1890
```

传输双方需要提前定义三种传感器检测结果的排列顺序，如温度传感器结果在前，湿度传感器在后，光照传感器位于最后。在上面的示例中，温度检测结果为 12.3，湿度检测结果为 80.5，光照检测结果为 1890。虽然文本类型比较实用，但没有 JSON 或 CBOR 类型灵活。

在某些物联网应用中，TLV 形式的二进制负载也广受欢迎，TLV 中的 T 表示类型，L 代表长度，而 V 表示具体内容，表面来看 TLV 形式的二进制类型非常节约空间，但是也带来一定的浪费或麻烦。

- T 的定义可能与 CoAP 中的方法重复，例如 T=0x01 表示数据上传，T=0x02 表示数据更新，那么此时 T 的定义便和 CoAP 中的 POST 方法和 DELETE 方法重复。
- V 的设计也存在很多“误区”，如整数传输的大小端问题、浮点数长度问题和大小端问题、负数保存的问题等。

5.8.3　JSON 类型

JSON（JavaScript Object Notation）[⊖]是一种轻量级的数据交换格式。JSON 格式易于人类阅读和编写，同时也易于机器解析和生成。JSON 采用完全独立于计算机语言的文本格式，但也使用了类似于 C 语言家族的习惯。这些特性使 JSON 成为理想的数据交换手段。JSON 有两种构成结构——JSON 对象和 JSON 数组。

1. JSON 对象

JSON 对象是一个无序的键值对集合。一个 JSON 对象以“{”开始并以“}”结束。每个“键”后跟一个“:”(分号)，每组键值对之间使用“,”(逗号）分隔。如图 5-21 所示为 JSON 对象的基本构造方法。

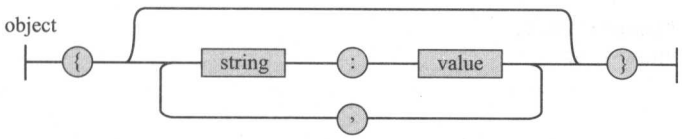


图 5-21　JSON 对象（图片来自 json.org）

2. JSON 数组

JSON 数组是值的有序集合。一个数组以“[”开始并以“]”结束。值之间使用“,”(逗号）分隔。如图 5-22 所示为 JSON 数组的基本构造方法。

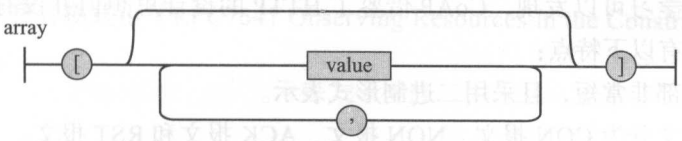


图 5-22　JSON 数组（图片来自 json.org）

在 JSON 数组中，value 可以为双引号包裹起来的字符串（string)、数值（number)、true、false、null。除此之外，值也可以是另一个 JSON 对象或数组。换句话说，JSON 对象和 JSON 数组可以任意嵌套。

3. JSON 格式示例

下面通过一个示例说明 JSON 对象和 JSON 数组的具体使用方法。该示例存在 JSON 数组和 JSON 对象嵌套情况，需要逐层由外向内分析，最外层的 JSON 对象包括两个键值对，第一个键值对的键名为“total”而键值为整数 3，在 JSON 格式中字符串必须使用引号（" "）包裹，而数值、true、false 和 null 不需要引号包裹；另一个键值对的键名为“rows”，而键值为一个 JSON 数组。在嵌套的 JSON 数组中包含三个类似的 JSON 对象。最内层的 JSON

⊖ http://json.org/json-zh.html。

对象包含两组键值对：第一组键值对的键名为“temp”，键值为24.5；第二组键值对的键名为“hum”，键值为89。

```
{
    "total": 3,
    "rows": [
        {
            "temp": 24.5,
            "hum": 89
        },
        {
            "temp": 25.5,
            "hum": 90
        },
        {
            "temp": 25.1,
            "hum": 92
        }
    ]
}
```

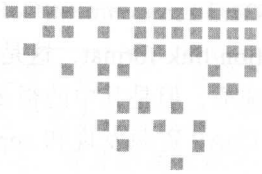
在互联网或物联网领域，JSON格式非常实用，所以有必要熟练掌握它。

5.9 本章小结

通过本章的学习可以发现，CoAP借鉴了HTTP的设计思想但并没有盲目抄袭或压缩HTTP。CoAP具有以下特点：

- ❑ CoAP首部非常短，且采用二进制形式表示。
- ❑ CoAP报文分为CON报文、NON报文、ACK报文和RST报文。
- ❑ COAP请求/响应可采用携带模式或分离模式。
- ❑ CoAP同样使用URI定位资源。
- ❑ CoAP具有和HTTP语义相似的请求方法和响应码。
- ❑ CoAP支持多种媒体类型。

在第7章，我们将使用C语言、Python、Java和Node.js语言实现CoAP客户端和服务器，通过后面章节的学习，我们会发现CoAP不但设计简洁，而且使用也非常方便。



第 6 章 Chapter 6

CoAP 扩展

6.1 本章主要内容

本章继续学习 CoAP 的其他部分，这些部分包括 CoAP 资源描述、CoAP 观察者模式。CoAP 资源描述由《RFC6690 Constrained RESTful Environments（CoRE）Link Format[⊖]》定义，而 CoAP 观察者模式由《RFC7641 Observing Resources in the Constrained Application Protocol[⊜]》定义。

6.2 CoAP 资源描述

CoAP 专门为设备间通信而设计，在设备通信过程中需要尽可能减少人为干预。为了实现设备在没有人干预的情况下正常工作，CoAP 引入了资源发现机制。这种机制可以帮助 CoAP 客户端理解 CoAP 服务器有哪些 URI 已经被支持，并且 CoAP 客户端可以理解这些 URI 的具体含义。CoAP 建议 CoAP 服务器总是支持一个 /.well-known/core 路由，该 URI 可以被任意 CoAP 客户端访问。

6.2.1 CoAP 资源描述原理

当 CoAP 客户端请求预先协商好的 /.well-known/core 路由时，CoAP 服务器将返回一系列的 URI 集合。这些 URI 集合遵循 CoRE 链接标准。CoRE 链接标准参考《RFC6690

⊖ https://datatracker.ietf.org/doc/rfc6690/。

⊜ https://datatracker.ietf.org/doc/rfc7641/。

Constraint RESTful Environment(CoRE) Link Format》。参考 CoRE 标准返回的列表类型被描述为 application/link-format，这是 CoAP 引入的一种全新的媒体类型。CoRE 链接标准定义了非常多的选项，但是其中的很多部分在实际使用中并不常见。下面我们通过一个简单的例子来说明 CoAP 资源发现和 application/link-format 媒体类型，这个示例的工作流程如图 6-1 所示。

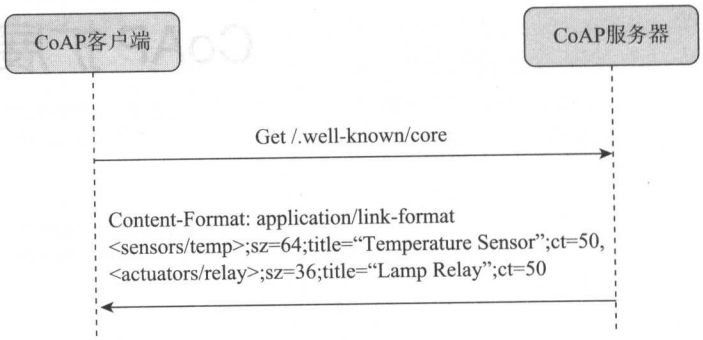


图 6-1　CoAP 资源发现过程

CoAP 客户端访问 /.well-known/core 路由之后获得的响应内容如下：

```
<sensors/temp>;sz=64;title="Temperature Sensor";ct=50,
<actuators/relay>;sz=32;title="Lamp Relay";ct=50
```

CoAP 服务器告诉 CoAP 客户端自身有两个资源（“sensors/temp”和“actuators/relay”）可以使用，通过以上例子可以获得以下有用的信息：

1）访问 /.well-known/core 路由只能通过 GET 方法，也就是说 /.well-known/core 路由不支持 POST、PUT 和 DELETE 方法。

2）CoAP 服务器有两个可以被其他设备访问的资源——“sensor/temp”和“actuators/relay”，资源之间采用“,”（逗号）分隔，每个资源都有各自不同的属性，这些属性通过“;”（分号）分隔。

3）CoAP 服务中存在一个传感器设备，该设备的 URI 为 sensors/temp，在 link-format 格式中 URI 被“<>”包裹；该资源还包括一个 title 属性，通过 title 属性可获知该设备的具体用途，此时该资源代表一个温度传感器；sz 属性代表该资源的分块大小，该属性在 CoAP 块传输中格外有用，此时的块大小为 64，也就是说当使用 GET 方法访问该资源时，最大数据包的长度为 64 字节；最后该资源还具有一个 ct 属性，此时 ct 属性的值为 50，该属性表示响应负载的媒体类型。在 CoAP 媒体类型一节可以获知，“50”代表 application/json。

4）CoAP 服务器中还存在一个可执行设备，该设备的 URI 为 actuators/relay；与传感器资源相似，该资源也包括一个 title 属性，通过 title 属性可以获知该设备是一个控制台灯电源的继电器设备；若使用 GET 方法或 POST 方法访问该资源时，最大数据包的长度为 32 字

节；与传感器资源相同，返回至CoAP客户端的负载媒体类型同样为application/json格式。

注意 link-format格式和JSON格式都使用了“,”（逗号）和“;”（分号）作为分隔符，JSON对象之间采用“;”作为分隔符，JSON数组中元素之前采用“,”作为分隔符。而link-format中，资源之间采用“,”作为分隔符，资源的属性描述之间采用“;”作为描述符。请读者注意其中的区别与联系。

6.2.2 CoAP资源描述详解

通过上面的例子已经对CoAP资源描述有一个大体的印象，下面再对资源“URI”、资源类型“rt”、接口说明“if”、负载类型“ct”、负载长度“sz”和可被观察“obs”做详细说明。

1. 资源URI

每一个资源必须包括URI，该URI使用“<>”包裹，例如上文示例中的<sensors/temp>和<actuators/relay>。

2. 资源类型rt

资源类型rt是“Resource Type”的简写，采用“键名称=键值”这样的方式描述。资源类型rt可以理解为一个与应用直接相关的描述字符串，它是用于描述该资源的特定名词。例如rt="block"，rt="Type1 Type2"。资源类型和资源title并不能完全等同，title更方便于人类阅读理解，而资源类型rt更像一个标记。

3. 负载类型ct

负载类型ct是“Content Format”的简写，采用“键名称=键值”这样的方式描述。该属性用于指示CoAP响应负载中的媒体格式。此处的键值采用十进制ASCII码表示，该值的范围必须在0～65 535之间，也就是说此处的键值为一个无符号16位整数。例如，ct=40表示负载媒体类型为application/link-format类型，ct=50表示负载媒体类型为application/json类型。

4. 负载长度sz

负载长度sz是“Maxium Size Estimae”的简写，也采用“键名称=键值”这样的方式描述。当使用GET方式或其他方式操作资源时，负载长度sz用于指示最大数据包长度。CoAP协议支持块传输，该参数对于需要块传输的应用来说非常有用。例如sz=128表示数据包的最大长度为128。在一些由IEEE 802.15.4组成的网络中，单个数据包的最大长度为127字节，即使通过6LoWPAN这样的头压缩技术，CoAP层可使用的有效负载长度也仅为80字节左右，在这些物联网应用中sz可以等于36或64。

5. 可被观察obs

可被观察obs是“observable”的简写。CoAP支持观察者模式，该模式与互联网应用

中的订阅 / 发布模式较为相似。若希望上文示例中的温度传感器资源也支持观察模式，可在该资源描述中加入 obs 属性，obs 属性只有键名称而没有键值。例如：

```
<sensors/temp>;sz=64;title="Temperature Sensor";ct=50,obs
```

6.3 CoAP 观察者模式

在物联网应用监控温度或湿度传感器是一个非常常见的需求，为了更好地满足这类需求 CoAP 引入了观察者模式。CoAP 客户端可以发送一个特殊观察请求到 CoAP 服务器。从该时间点开始计算，服务器将保存客户端的连接信息，一旦温度发生变化，服务器将会把新结果反馈至客户端。如果 CoAP 客户端不再希望获得温度检测结果，那么 CoAP 客户端将会发送一个注销请求或 RST 复位请求，此时 CoAP 服务器便会清除与客户端的连接信息。

6.3.1 观察者模式原理

在互联网应用中浏览器为了实时获取某个资源，可通过周期性发送 Ajax 请求的方式，这样的方式称为 Ajax 轮询。在物联网应用中，监控温度或湿度传感器的变化情况也可以采用轮询的方式，轮询过程如图 6-2 所示。

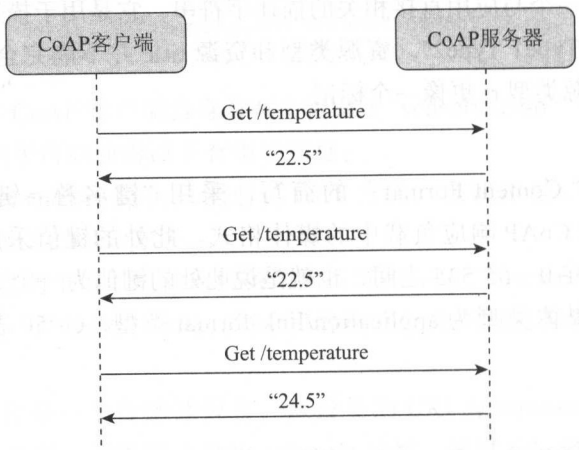


图 6-2 CoAP 轮询传感器检测结果

如图 6-2 所示的监控过程是非常低效的，这种低效主要体现在两个方面。第一，CoAP 客户端为了获取服务器的传感器资源，每次都需要发送一次 GET 请求，而每次 GET 请求的内容完全相同，这显然是一种浪费；第二，CoAP 客户端按照一定的时间间隔请求内容，而该时间间隔可能与服务器的传感器检测更新时间间隔并不匹配，这将会造成多次请求但是获得同一个结果的糟糕情况。为了避免这种情况，CoAP 中引入了观察者模式，观察者模式的工作过程如图 6-3 所示。

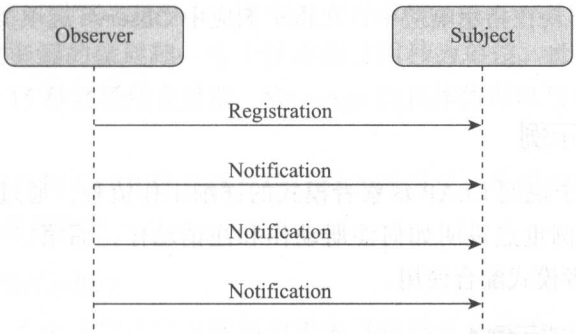


图 6-3　CoAP 观察者模式原理

CoAP 观察者模式引入了 4 个基本概念——Observer、Subject、Registration 和 Notification。

- Observer：CoAP 观察者是监控某种资源变化的 CoAP 客户端。
- Subject：CoAP 主题是 CoAP 服务器中某个具体资源，该资源将会随着时间的变化频繁更新。CoAP 主题正是 CoAP 观察者感兴趣的部分。
- Registration：CoAP 注册是一个特殊的 CoAP GET 请求，该 GET 请求告知服务器此时 CoAP 客户端试图获取哪个资源的实时状态。服务器收到这个特殊的 GET 请求之后，将把 CoAP 客户端的信息保存到资源观察者列表中。
- Notification：CoAP 指示，一旦服务的资源发生变化，CoAP 服务器将会根据观察者列表中记录的信息，向 CoAP 客户端反馈资源的当前状态。

在 CoAP 观察者模式中，CoAP 客户端作为观察者，而 CoAP 服务器作为被观察者，观察过程总是从一个特殊的观察 GET 请求开始。

6.3.2　CoAP 观察选项

为了实现观察与通知功能，CoAP 在选项中增加了 Observe 选项，Observer 选项见表 6-1。

表 6-1　Observe 选项说明

选 项 值	选 项 名 称	数 据 类 型	长 度 说 明
6	Observe	uint	0～3

当 GET 请求中出现 Observe 参数时，CoAP 服务器并不会向 CoAP 客户端反馈请求 URI 中指定的资源，而是把 CoAP 客户端和被订阅的资源信息保存到观察者列表中。在 GET 请求中，Observe 有两个可能的取值——0 或 1。

- Observe=0，表示注册操作，如果 CoAP 客户端信息并不在观察者列表中，那么服务器收到该请求之后会把客户端信息插入到观察者列表中。
- Observe=1，表示注销操作，CoAP 服务器将会把观察者列表中的客户端信息删除，这也就意味着该订阅过程的结束。

Observe 选项也出现在指示响应中，在指示响应中 Observe 选项类似于序号，该序号将会不断增加。

6.3.3 观察者模式示例

下面通过两个例子说明 CoAP 观察者模式的详细工作流程，通过示例加深对观察者模式的理解。第一个示例重点说明如何注册动作和注销动作，而第二个示例重点说明 Max-Age 选项如何与观察者模式配合使用。

1. CoAP 观察者模式示例 1

如图 6-4 所示是一个非常完整的 CoAP 观察者模式工作流程，CoAP 客户端订阅 CoAP 服务器中的温度传感器资源，一段时间之后 CoAP 客户端取消了订阅，整个资源观察过程也就此结束。在该示例中 CoAP 客户端和服务器都正常工作未出现数据包丢失等异常情况。

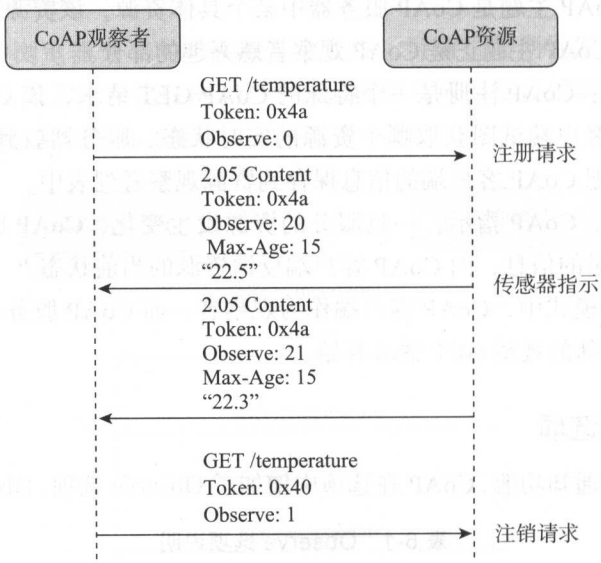


图 6-4 CoAP 资源发现示例 1

1）CoAP 客户端向 CoAP 服务器订阅温度传感器资源。CoAP 客户端发送带有 Observe 选项的 GET 请求，此时 Observe=0，表示订阅注册动作。

2）CoAP 发送订阅请求时，还可以在请求中加入 Token，此时的 Token 长度为 1 字节，值为 0x4a。通过第 5 章可以获知 Token 与应用直接相关，那么在 CoAP 观察过程中 Token 值应保持不变。

3）CoAP 服务器同意 CoAP 客户端的资源注册请求，向客户端返回温度传感器检测结果。响应首部中包含“2.05 Content”响应码。除此之外响应首部中还包括 Observe 选项，Observe 选项好比订阅过程的“计数器”，在每个 CoAP 响应中其值逐渐累加。

4）Max-Age 选项也经常出现在 CoAP 观察者模式中。在 CoAP 中资源也有“新鲜”的概念，Max-Age 好比资源的保质期，这个保质期使用秒为单位。此处 Max-Age=15，表示温度传感器检测结果 15 秒之后便会过期。Max-Age 的具体值可以与传感器检测结果的更新周期一致。

5）CoAP 客户端可向 CoAP 服务器发送订阅注销请求，此时 Observe=1 表示 CoAP 客户端不再关心温度传感器资源。

2. CoAP 观察者模式示例 2

在上一个示例中传输过程并没有数据包丢失这样的异常情况，一旦出现数据包丢失，CoAP 观察者模式也可以借助 Max-Age 选项从这种错误中尽快恢复，如图 6-5 所示。

1）CoAP 资源注册过程与示例 1 相同。

2）CoAP 服务器响应中包含 Max-Age=15，表示温度传感器资源的“保质期”只有 15 秒。CoAP 客户端收到温度传感器资源之后便开始计时，一般情况下在 0 到 Max-Age 之间便可收到下一次传感器指示。但是由于某种原因 CoAP 服务器返回的 CoAP 响应在传输过程中丢失。

3）Max-Age 超时时间已经到达，传感器数据结果已经不再“新鲜”。CoAP 客户端认为数据传输过程发生异常，为了从这种异常状态下恢复，CoAP 客户端重新发送了资源注册请求。Max-Age 选项使 CoAP 客户端从错误状态中及时恢复。

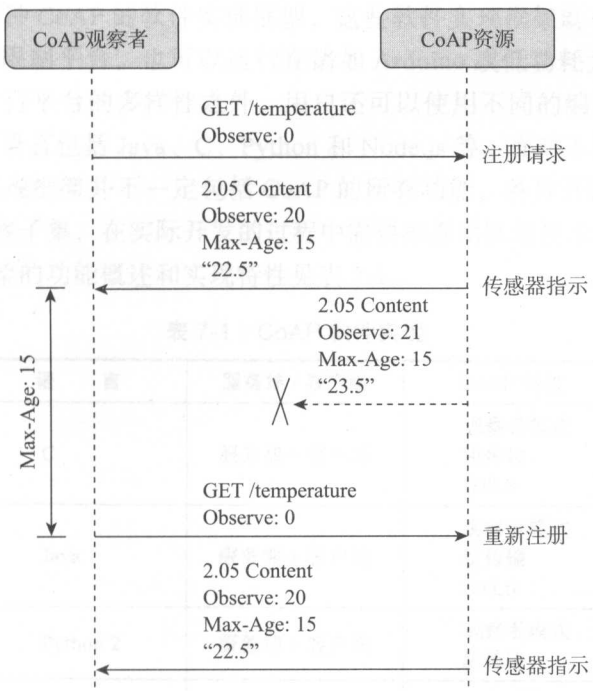


图 6-5　CoAP 资源发现示例 2

6.4 本章小结

本章分析了 CoAP 的两个扩展机制——CoAP 资源描述和 CoAP 观察者模式。CoAP 主要为设备间通信服务，CoAP 资源描述部分可帮助 CoAP 客户端了解 CoAP 服务器中包含哪些资源，CoAP 客户端应该如何理解和利用这些资源。CoAP 是一种典型的物联网协议，它必须要符合物联网应用中的常见需求。在物联网应用中获取传感器的检测结果是一个最为常见的需求，为 CoAP 中增加了观察者模式后，终端可通过订阅的方式获取某个传感器的检测结果，而不需要使用周期性轮询这样的低效方案。通过资源描述和观察者模式，我们可以发现 CoAP 为物联网而生，为了更好地符合物联网应用 CoAP 将会增加更多更实用的扩展部分。

第 7 章 Chapter 7

CoAP 软件实现

7.1 本章主要内容

本章将介绍多种 CoAP 的软件实现框架。随着 CoAP 标准的完善和开源社区的不断努力，市面上出现了多种 CoAP 的软件实现框架，这些软件实现框架既可以运行在 Windows 或 Linux 平台等非受限制平台，也可以运行在诸如 Arduino 或低功耗无线传感网终端等受限制设备中。除了运行平台的多样性之外，用户还可以使用不同的编程语言实现 CoAP 的各种功能，这些编程语言包括 Java、C、Python 和 Node.js 等。面对不同的平台与不同的使用场景，各种开源实现框架并不一定包括 CoAP 的所有功能，各种开源实现框架往往只是 CoAP 众多标准的一些子集，在实际开发的过程中需要根据团队的技术偏好和具体需求灵活选择。CoAP 实现框架的功能概述和实现特性见表 7-1。

表 7-1 CoAP 软件实现

名　称	语　言	服务端 / 客户端	CoAP 特性	许 可 证
libcoap	C	服务端 + 客户端	观察者模式 块传输 DTLS	BSD/GPL
Californium	Java	服务端 + 客户端	观察者模式 块传输 DTLS	EPL+EDL
txThings	Python 2	服务端 + 客户端	观察者模式 块传输	MIT
aiocoap	Python 3	服务端 + 客户端	观察者模式 块传输	MIT

（续）

名　称	语　言	服务端 / 客户端	CoAP 特性	许 可 证
node-coap	Node.js	服务端 + 客户端	观察者模式 块传输	MIT
Erbium	C	服务端 + 客户端	观察者模式 块传输 DTLS	BSD

本章节相关示例代码均位于 GitHub 代码仓库中，用户可以使用 git clone 指令复制示例代码仓库，代码仓库的地址如下：

https://github.com/xukai871105/the_beginning_of_coap

其中：

1）californium 入门示例相关文件位于 simple_demo/cf_demo 目录中。

2）aiocoap 入门示例相关文件位于 simple_demo/aiocoap_demo 目录中。

3）node-coap 入门示例相关文件位于 simple_demo/nodecoap_demo 目录中。

7.2 libcoap

与 Linux 平台大多数以 lib 开头的工具一样，libcoap 是一款简单实用但功能完整的开发工具。libcoap 不但提供了一个实用的 coap-client 命令行工具，也提供了一个用于测试目的的 coap-server 命令行工具。

libcoap 提供一个动态链接库 libcoap.so 文件，用户可以使用 libcoap 提供的 API 实现各种形式的 coap-server 或 coap-client。libcoap 还是一款非常高效的命令行调试工具，libcoap 提供的 coap-client 工具相当于 HTTP 领域中的 cURL 工具，该工具可以实现各种各样的 CoAP 请求。更多信息可前往 libcoap 的官方网址和 github 代码仓库获得。

- libcoap 官方网址：https://libcoap.net/
- libcoap 代码仓库：https://github.com/obgm/libcoap

libcoap 是一个多功能的工具，虽然使用其他的脚本语言例如 Python 和 Node.js 也可以编写出功能相同的 CoAP 请求，但 libcoap 工具的使用更加方便灵活。

7.2.1 libcoap 安装

libcoap 的安装过程大致可分为安装依赖项、获取源代码、编译与安装这 4 个步骤。可通过多种方式获取 libcoap 的源代码，若在 Linux 主机中已经安装 Git 工具，可通过“git clone”方式获得 libcoap 最新代码；若尚未安装 git 工具，也可直接下载 libcoap 官方提供的稳定版源代码。

下面将详细说明如何在树莓派 3 代中通过源代码的方式安装 libcoap，其他 Linux 主机

中安装 libcoap 的步骤与树莓派 3 代中安装 libcoap 的步骤几乎相同。

1. 安装依赖项

通过源代码方式安装 libcoap 时需要提前安装 automake、libtool 和 doxygen 等依赖工具，在树莓派 3 代中可通过 apt-get 工具安装以上依赖工具。

```
# 安装依赖工具之前，请先使用apt-get update指令更新软件源
sudo apt-get update
# 必选依赖项 automake autoconf
sudo apt-get install automake
sudo apt-get install autoconf
# 可选依赖项
sudo apt-get install libtool
sudo apt-get install doxygen
sudo apt-get install asciidoc
sudo apt-get install cunit
```

2. 获取源代码

通过 git 工具获取最新 libcoap 源代码。

```
# 创建一个文件夹，用于保存libcoap源代码
mkdir -p software
# 进入software目录
cd software
# 复制源代码
git clone https://github.com/obgm/libcoap
```

3. 编译与安装

libcoap 安装过程也相对简单，但具体过程中也会需要处理一些问题，需要根据错误提示耐心修正。一般来说 libcoap 的源代码安装过程可分为以下几个步骤：

1）./autogen.sh 生成 configure 可执行文件。

2）./configure 配置软件环境。

3）make 编译源代码。

4）sudo make install 把相关头文件安装至 /usr/local/include/libcoap，把相关库文件安装至 /usr/local/lib。

5）sudo ldconfig 重新搜索当前系统中的动态链接库。

在控制台中依次输入以下指令便可完成 libcoap 的安装。

```
# 进入libcoap源代码所在目录
cd libcoap
# 生成构建脚本
./autogen.sh
# 配置软件环境
./configure --disable-documentation
# 编译libcoap源代码
```

```
make
# 执行安装过程
sudo make install
# 重新搜索系统中的动态链接库
sudo ldconfig
```

在执行 configure 脚本时可输入不同的参数。

- --enable-tests 生成单元测试用例，若启用该参数需要在主机中提前安装 cunit 测试框架。
- --enable-examples 生成符合 POSIX 标准的示例，默认使能了该参数。
- --enable-documentation 生成说明文档，若启用该选项需要在主机中提前安装 doxygen 和 a2x 工具。默认设置中已经使能该参数，所以需要提前在主机中安装 doxygen 和 a2x 工具，否则可输入 ./configure --disable-documentation 以禁止生成说明文档。

7.2.2 libcoap 使用详解

libcoap 包含两个主要的命令行工具 coap-server 和 coap-client。coap-server 工具可快速搭建一个测试用途的 CoAP 服务器。coap-client 类似于 HTTP 中常用的 cURL 工具，通过 coap-client 可以设置 CoAP 首部中的各种选项，执行 CoAP GET 或 PUT 请求等。下面分 coap-server 工具和 coap-client 工具两部分说明如何在物联网系统开发实战中使用 libcoap 工具。

1. coap-server

（1）coap-server 用法说明

```
coap-server [-A address] [-p port]
```

（2）coap-server 参数说明

coap-server 的参数说明见表 7-2。

表 7-2 libcoap coap-server 工具参数说明

参 数	参数说明
-A address	一般情况下 Linux 主机有多个 IP 地址，可通过该参数绑定其中的一个 IP 地址。该 IP 地址可以是 IPv4 回环地址 127.0.0.1，也可以是 IPv6 形式的回环地址 ::1，更可以是本机的 IPv4 地址，例如 192.168.0.103，还可以是本机的某个 IPv6 地址，例如 2020:CA28:0:0:23:222:0:2900
-p port	指定侦听端口号，CoAP 协议默认端口号为 5683，CoAP 加密通信的默认端口号为 5684
-v num	设置调试等级，默认为 3 级

（3）coap-server 用法示例

下面通过几个示例说明 coap-server 工具的使用方法。

```
# 绑定本地IPv4回环地址
coap-server -A 127.0.0.1
# 绑定本地IPv6回环地址
coap-server -A ::1
# 绑定本地IPv4地址，并指定端口号
coap-server -A 192.168.0.103 -p 5683
# 绑定本地IPv6地址
./coap-server -A 2020:CA28:0:0:23:222:0:2900
```

若已知 Linux 主机的 IPv4 地址，推荐使用“coap-server –A 192.168.0.103 –p 5683”这样的写法，通过 -A 参数指定主机地址，通过 -p 参数指定服务端口号。

2. coap-client

（1）coap-client 用法说明

相比于 coap-server 工具，coap-client 工具的参数较多，coap-client 的具体使用方法如下：

```
coap-client [-A type...] [-t type] [-b [num,]size] [-B seconds] [-e text] [-m
method] [-N] [-o file] [-P addr[:port]] [-p port] [-s duration] [-O num,text] [-T
string] [-v num] [-a addr]
 [-u user] [-k key] URI
```

（2）coap-client 参数说明

coap-client 的参数说明见表 7-3。

表 7-3　libcoap coap-client 工具参数说明

参　数	参数说明
URI	符合 CoAP 标准的 URI。例如 coap://wsncoap.org:5683/test
-A type	设置期望获得的媒体类型，该参数既可以是一个符合 CoAP 媒体类型标准的枚举值，也可以是相应的字符串描述 0　text/plain 40　application/link-format 41　application/link-format 42　application/octet-stream 47　application/exi 50　application/json 60　application/cbor 推荐在实际使用的过程中选择枚举值
-t type	指示 CoAP 负载中的媒体类型，和 -A 参数相同，该参数既可以是一个符合 CoAP 媒体类型标准的枚举值，也可以是相应的字符串描述 0　text/plain 40　application/link-format 41　application/link-format 42　application/octet-stream 47　application/exi 50　application/json 60　application/cbor 推荐在实际使用的过程中选择枚举值

（续）

参　数	参数说明
-b [num,] size	设置 CoAP 分块传输大小。size 参数用于设置分块大小，可选值 16、32、64、128、256、512 和 1024 等；num 参数是一个可选参数，该参数用于设置分块传输的序号偏移量
-B seconds	设置 CoAP 等待响应的超时时间，单位为秒
-e text	设置 CoAP 请求负载，负载类型必须为文本类型
-f file	把指定文件作为 PUT 或 POST 方法的负载，使用 "-" 表示 STDIN（标准输入）
-m method	指定 CoAP 请求方法，CoAP 请求方法包括 GET、PUT、POST 和 DELETE，默认的 CoAP 请求方法为 GET
-N	发送 NON 请求
-o file	把 CoAP 响应写入到指定文件中，使用 "-" 表示 STDOUT（标准输出）
-p port	指定 coap client 侦听端口。设置 coap client 端口之前需要确定该端口并没有被其他应用或服务占用。该参数为设置 CoAP 客户端发送 UDP 数据时使用的端口号，请不要与 CoAP 服务器中的端口号相混淆。若需指定服务器的端口号可在 URI 参数中指定
-s duration	启用 CoAP 观察者模式使用，duration 用于设置观察模式开始到观察者模式结束的时间间隔，单位为秒。例如，-s 10 表示 CoAP 客户端观察某个资源 10 秒，10 秒之后立刻结束观察。请注意被观察者将源源不断地向 CoAP 观察者提供数据，数据之间的时间间隔由被观察者决定，与此处的 -s duration 没有任何关系
-v num	指定调试信息输出等级。默认为 3 级，最高为 9 级
-T token	设置用户自定义 CoAP Token
-k key	该功能位于 DTLS 分支，若使用加密 CoAP 功能必须提供该参数。 设置用户的预分享密钥 Pre-shared Key
-u user	该功能位于 DTLS 分支，若使用加密 CoAP 功能必须提供该参数。 设置用户标识符。由于 CoAP Server 中将会保存多个预分享密钥，每个预分享密钥可能属于不同的用户，CoAP Server 通过该标识符确定具体用户的 Pre-shared Key

（3）coap-client 用法示例

下面通过几个具体示例说明 coap-client 的使用方法。

```
# 使用CoAP get方法访问本机（IPv4）.well-known/core知名路由
coap-client -m get coap://127.0.0.1/.well-known/core
# 使用CoAP get方法访问本机（IPv6）.well-known/core知名路由
coap-client -m get coap://[::1]/.well-known/core
# 使用CoAP get方法访问192.168.0.6主机中time资源
coap-client -m get -T cafe coap://192.168.1.103/time
# 使用CoAP put方法更新192.168.1.103主机中的time资源
echo 1000 | coap-client -m put -T cafe coap://192.168.0.6/time -f -
```

虽然 coap-client 的参数较多，但使用过程并不复杂。首先，需要指定 CoAP 请求方法，通过 -m 参数便可设定 CoAP 请求方法，CoAP 请求方法包括 GET、PUT、POST 和 DELETE 四种；接着，若需要指定 CoAP 请求负载，可使用 -e 参数设置请求负载，请求负载的类型为文本类型；最后指定 CoAP 资源 URI，coap-client 工具支持 IPv6 地址，如果使用 IPv6 形

式的主机地址，那么 IPv6 地址需要包裹在中括号中，如 coap://[fd00::212:4b00:42a:fd00]/time；如果 URI 还包括非默认端口号，请注意一定不能把端口号也放在中括号中，如下的 CoAP URI 才是正确的：coap://[fd00::212:4b00:42a:fd00]:5684/time。

7.2.3　libcoap 入门示例

下面结合树莓派 3 代给出一个 libcoap 入门示例，通过一个具体的示例掌握 libcoap 的使用方法。在入门示例中包含两个 Linux 设备——Linux 主机和树莓派，Linux 主机作为 CoAP 客户端，而树莓派 3 代作为 CoAP 服务器，libcoap 入门示例的网络结构如图 7-1 所示。在 CoAP 服务器中提供一个访问系统时间的路由，使用树莓派的系统时间是因为它是一个不断变化的参数，也就是说每次 CoAP 请求都可以获得不一样的 CoAP 响应。通过这种变化可帮助用户更好地了解 CoAP 本身，了解在实际的应用中哪些细节固定不变，哪些细节将不停地发生变化。libcoap 入门示例中将会使用两组不同的 coap-client 指令：

1）获得 CoAP 服务器公开信息 coap-client -m get coap://192.168.0.6/.well-known/core。

2）获得 CoAP 服务器系统时间 coap-client -m get coap:// 192.168.0.6 /time。

3）启动观察者模式，观察 CoAP 服务器系统时间 coap-client -m get –s 100 coap:// 192.168.0.6 /time。

为了成功运行 libcoap 入门示例，需要在 Linux PC 主机和树莓派中正确安装 libcoap。

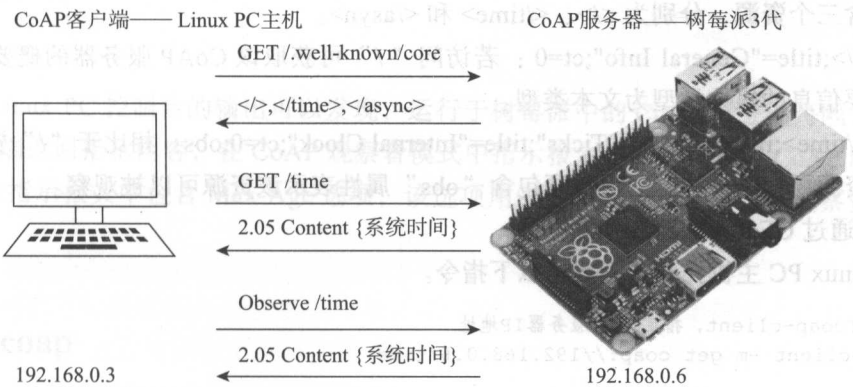


图 7-1　libcoap 入门示例

1. 服务器实现

首先在树莓派中运行 coap-server 工具。coap-server 包括一个 time 资源，通过 GET 方法或 Observer 方法访问该资源可获取树莓派的系统时间。

```
# 运行coap-server，并指定树莓派的IP地址
coap-server -A 192.168.0.6
```

运行 coap-server 时，-A 参数用于指定 Linux 主机 IP 地址也就是树莓派的 IP 地址，本例中树莓派的 IPv4 地址为 192.168.0.6。若不清楚树莓派的 IP 地址，可在树莓派中使用 ifconfig 指令查看。

2. 客户端实现

（1）访问 /.well-known/core

在 Linux PC 主机中重新打开一个控制台，在控制台中输入以下命令。

```
# 运行coap-client，指定CoAP Server IP地址
coap-client -m get coap://192.168.0.6/.well-known/core
```

- -m get 通过 -m 参数指定访问该资源的 CoAP 方法，此处为 GET 方法。
- 访问 URI 为 coap://192.168.0.6/.well-known/core。

Linux PC 主机将获取以下内容：

```
v:1 t:CON c:GET i:2836 {} [ ]
</>;title="General Info";ct=0,
</time>;if="clock";rt="Ticks";title="Internal Clock";ct=0;obs,
</async>;ct=0
```

- v:1 t:CON c:GET i:2836 给出 CoAP 请求的首部信息，v:1 代表 CoAP 版本编号，无论如何 CoAP 的版本编号均为 1；t:CON 表示 CoAP 请求为 CON 类型请求；c:GET 表示请求方法为 GET 请求；i:2836 表示报文序号为 2836。
- 访问 .well-known/core 可获取服务器中所有资源的提示信息，该 CoAP 服务器中包含三个资源，分别为 </>、</time> 和 </asyn>。
- </>;title="General Info";ct=0：若访问“/”可获取该 CoAP 服务器的概要信息，概要信息的媒体类型为文本类型。
- </time>;if="clock";rt="Ticks";title="Internal Clock";ct=0;obs：相比于“/”资源，time 资源要复杂一些，time 资源包含“obs”属性表示该资源可以被观察。

（2）通过 GET 方法获取系统时间

在 Linux PC 主机控制台中运行以下指令。

```
# 运行coap-client，指定CoAP服务器IP地址
coap-client -m get coap://192.168.0.6/time
```

- -m get 通过 -m 参数指定访问该资源的 CoAP 方法，此处仍为 GET 方法。
- 访问 URI 为 coap://192.168.0.6/time。

运行于树莓派 3 代中的 CoAP 服务器将会返回以下内容。

```
v:1 t:CON c:GET i:4b5d {} [ ]
Aug 01 06:07:02
```

- v:1 t:CON c:GET i:4b5d 给出了 CoAP 请求的首部信息，v:1 代表 CoAP 版本编号；t:CON 表示 CoAP 请求为 CON 类型请求；c:GET 表示请求方法为 GET 请求；i: 4b5d

表示报文编号为 4b5d。

- Aug 01 06:07:02 表示此时树莓派 3 的系统时间。

由于 time 服务中并没有考虑时区和时间表示方式等因素，所以 CoAP 响应中树莓派的系统时间并不一定与当前时间完全吻合，但这并没有太大的关系，此处仅仅是为了说明 libcoap 可以在 Linux 主机和树莓派中正常工作。

（3）通过 Observer 观察系统时间

通过 Observer 观察 CoAP 服务器中的 time 资源，在 Linux PC 控制台输入以下指令：

```
# 运行coap-client，指定CoAP Server IP地址
coap-client -m get -s 100 coap://192.168.0.6/time
```

- -m get 通过 -m 参数指定访问该资源的 CoAP 方法，CoAP 中 Observer 也可以理解为一种特殊的 GET 方法，但在 libcoap 中执行 Observer 一定要在 coap-client 指令中增加 -s 参数。
- –s 100 表示持续观察 time 资源 100 秒，在这 100 秒的时间内 CoAP 服务器将向 CoAP 客户端源源不断地返回树莓派 3 代内部的系统时间。
- 访问 URI 为：coap://192.168.0.6/time。

Linux PC 控制台将获得以下内容：

```
v:1 t:CON c:GET i:cb5a {} [ ]
Dec 10 08:39:54
Dec 10 08:39:56
# 省略若干内容
Dec 10 08:41:20
Dec 10 08:41:22
```

通过 Linux PC 控制台的输出可以发现，运行于树莓派中的 CoAP 服务器每隔 2 秒钟向 CoAP 客户端返回指示内容，在 CoAP 观察者模式中指示报文的时间间隔由 CoAP 服务器决定，一般在指示报文中包含 Max-Age 选项，该选项用于告知 CoAP 客户端被观察资源的更新周期。

7.3　aiocoap

Python 是一款非常容易学习的解释性语言，用户编写脚本可以直接被运行，而不需要先整体编译成机器代码。此外 Python 安装和运行环境设置也非常容易，开发者社区相对活跃。Python 2 和 Python 3 也是开源硬件树莓派的默认安装软件。

在 Python 领域中也有不少 CoAP 的实现框架，其中 txThing [⊖]和 aiocoap [⊜]使用较为广泛。

⊖ https://github.com/mwasilak/txThings。

⊜ https://github.com/chrysn/aiocoap。

txThing 依赖 Python 2.7，采用 Twisted 框架实现。而 aiocoap 采用 Python 3 开发，使用 Python 3.4 版本之后自带的异步 IO 框架 asyncio 实现。aiocoap 和 txthing 有继承关系，aiocoap 继承了 txThing 的绝大部分功能，并把异步 IO 框架由 Twisted 改为 Python 3.4 之后自带的 asyncio。

7.3.1 aiocoap 安装

1. 源代码方式安装 Python 3.5

aiocoap 入门示例依赖 Python 3.5，所以运行本节入门示例之前需在树莓派中安装 Python 3.5 或更高版本，下面说明如何通过源代码方式安装 Python 3.5.2，在树莓派中新建控制台，在控制台中依次输入以下指令：

```
# 获取Python 3.5源代码
wget https://www.python.org/ftp/python/3.5.2/Python-3.5.2.tgz
# 解压Python 3.5源代码
tar -zxvf  Python-3.5.2.tgz
# 进入Python 3.5目录
cd Python-3.5.2/
# 生成makefile文件
./configure
# 编译源代码
make
# 安装Python3.5
sudo make install
```

2. 通过 pip 工具安装 aiocoap

pip 工具是 Python 3 的默认包管理工具，可通过 pip 工具安装 aiocoap。在树莓派控制台中依次输入以下指令便可完成 aiocoap 的安装。

```
# 通过pip3安装aiocoap
sudo pip3 install aiocoap
# 通过pip3安装aiocoap的依赖项LinkHeader
sudo pip3 install LinkHeader
```

由于树莓派中已经安装了 Python 2，所以树莓派中已经包含了两个不同版本的 pip 工具。如果需要把扩展库安装到 Python 3 的 lib 目录中需要使用 pip3 工具，若直接使用 pip 工具那么扩展库将被安装至 Python 2 的 lib 目录中，此时将无法使用本节相关的 aiocoap 示例。除了安装 aiocoap 之外，还需要安装 aiocoap 的依赖库 LinkHeader。若已经安装过 aiocoap，可通过 upgrade 指令更新 aiocoap，在控制台中输入以下指令便可完成更新操作：

```
# 通过pip3工具更新aiocoap
sudo pip3 install aiocoap --upgrade
```

7.3.2 aiocoap 入门示例

下面通过两个示例说明如何使用 aiocoap。入门示例中包括两台设备，这两台设备分别扮演 CoAP 客户端和 CoAP 服务器的角色，此处树莓派 3 代作为 CoAP 服务器，另一台

Linux PC 主机作为 CoAP 客户端。在运行入门示例之前需要保证树莓派和 Linux PC 主机都已经正确安装合适版本 Python 3，而且 Linux 主机可以通过网络正常访问树莓派 3 代。入门示例中树莓派 3 代作为服务器将运行 test-server.py，而 Linux PC 主机作为服务器端将运行 test-client.py。树莓派中 CoAP 服务器提供一个系统时间服务（资源），通过 coap://192.168.0.6/time 便可访问该资源，CoAP 服务器中的系统时间资源仅支持 GET 方法。aiocoap 入门示例的大致工作流程如图 7-2 所示。

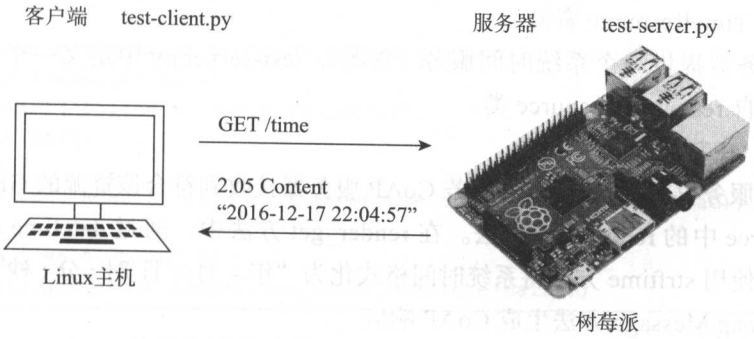


图 7-2　aiocoap 入门示例

1. 服务器实现

test-server.py 具体代码如下：

代码清单7-1　test-server.py

```
#!/usr/bin/env python3
import datetime
import logging
import asyncio

import aiocoap.resource as resource
import aiocoap

class TimeResource(resource.Resource):
    def __init__(self):
        super(TimeResource, self).__init__()

    async def render_get(self, request):
        payload =
datetime.datetime.now().strftime("%Y-%m-%d %H:%M:%S").encode('ascii')
        return aiocoap.Message(code=aiocoap.Code.CONTENT, payload=payload)

logging.basicConfig(level=logging.INFO)
logging.getLogger("coap-server").setLevel(logging.DEBUG)

def main():
    root = resource.Site()
```

```
    root.add_resource(('.well-known', 'core'),
resource.WKCResource(root.get_resources_as_linkheader))root.add_resource(('time',),
    TimeResource())
    asyncio.Task(aiocoap.Context.create_server_context(root))
    asyncio.get_event_loop().run_forever()

if __name__ == "__main__":
    main()
```

（1）定义 TimeResource 资源

CoAP 服务器提供一个系统时间服务（资源），test-server.py 中定义一个 TimeResource 类，该类继承自 resource.Resource 类。

（2）处理 GET 方法

系统时间服务仅支持 GET 方法，若 CoAP 服务器接收到符合该资源的 GET 请求，将调用 TimeResource 中的 render_get 方法。在 render_get 方法中，通过 datetime 取出树莓派内系统时间，并使用 strftime 方法将系统时间格式化为“年 - 月 - 日 时 : 分 : 秒”这样的形式，最后调用 aiocoap.Message 方法生成 CoAP 响应。

```
async def render_get(self, request):
    payload = datetime.datetime.now().strftime("%Y-%m-%d %H:%M:%S").encode('ascii')
    return aiocoap.Message(code=aiocoap.Code.CONTENT, payload=payload)
```

（3）增加 TimeResource 资源

完成 TimeResource 资源定义之后便可把该资源加入到服务器根站点中。除了加入 TimeResource 资源之外，test-server.py 还增加了 well-known 资源，well-known 资源的具体内容和处理方法将由 aiocoap 自动生成，并不需要编写额外的代码。

```
root.add_resource(('.well-known', 'core')
resource.WKCResource(root.get_resources_as_linkheader))
root.add_resource(('time',), TimeResource())
```

（4）循环处理 CoAP 请求

最后利用 python 3.4 之后自带的 asyncio 异步处理框架处理来自不同客户端的 CoAP 请求。

```
asyncio.Task(aiocoap.Context.create_server_context(root))
asyncio.get_event_loop().run_forever()
```

2. 客户端实现

test-client.py 具体代码如下：

代码清单7-2　test-client.py

```
#!/usr/bin/env python3

import logging
import asyncio
```

```
import aiocoap

logging.basicConfig(level=logging.INFO)

async def main():
    protocol = await aiocoap.Context.create_client_context()
    request = aiocoap.Message(code=aiocoap.Code.GET, uri='coap://
<raspberry_ip_address>/time')

    try:
        response = await protocol.request(request).response
    except Exception as e:
        print('Failed to fetch resource:')
        print(e)
    else:
        print('Result: %s\n%r' % (response.code, response.payload))

if __name__ == "__main__":
    asyncio.get_event_loop().run_until_complete(main())
```

下面说明 test-client 的具体工作流程。

（1）构造 CoAP 请求

构造 CoAP 请求时需要指定 CoAP 请求方法和 URI，test-client.py 中 CoAP 的请求方法为 GET 方法，URI 为 coap://<raspberry_ip_address>/time。请根据实际运行环境替换 <raspberry_ip_address> 为实际的树莓派 IPv4 地址。

```
request = aiocoap.Message(code=aiocoap.Code.GET, uri='coap://192.168.0.6/time')
```

（2）发送 CoAP 请求并输出结果

通过 protocol.request 方法发送构造 CoAP 请求，获得 CoAP 响应之后把响应码和响应负载通过 print 函数输出至控制台中。

```
try:
    response = await protocol.request(request).response
except Exception as e:
    print('Failed to fetch resource:')
    print(e)
else:
    print('Result: %s\n%r'%(response.code, response.payload))
```

3. 动手测试

完成 test-server.py 和 test-client.py 之后便可动手进行测试，在树莓派 3 代中运行 test-server.py，在另一台 Linux PC 主机中运行 test-client.py。

（1）运行 CoAP 服务器

在树莓派 3 代控制台中先运行 test-server.py。

```
python3 test-server.py
```

（2）获取 TimeResource 资源

然后在另一台 Linux PC 主机中运行 test-client.py。

```
python3 test-client.py
```

若 test-client.py 成功运行，那么控制台将会获得以下输出内容：

```
Result: 2.05 Content
b'2016-08-01 17:30:24'
```

2.05 Content 表示 CoAP 服务器成功响应了 CoAP 客户端的请求，CoAP 响应负载中包含树莓派 3 代的系统时间，系统时间采用“Y-m-d H:M:S”，也就是“年 - 月 - 日 时 : 分 : 秒”这样的格式。简单比较一下系统时间便可验证 CoAP 服务器是否工作正常。

7.3.3 aiocoap 块传输示例

在入门示例的 CoAP 服务中仅包含一个 TimeResource 资源，本节在 test-coap.py 的基础上再增加一个块（Block）资源。该资源的响应负载内容较长，CoAP 客户端可通过分块传输的方式获取该资源，分块的大小由 CoAP 客户端定义。如图 7-3 所示。

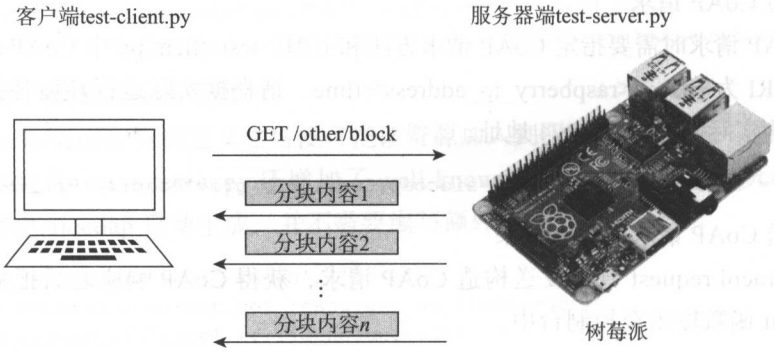


图 7-3 aiocoap 分块传输示例

1. 服务器实现

更改之后的 test-server.py 的具体代码如下：

代码清单7-3 更改之后的test-server.py

```
#!/usr/bin/env python3
import datetime
import logging
import asyncio

import aiocoap.resource as resource
import aiocoap

class TimeResource(resource.Resource):
```

```
    def __init__(self):
        super(TimeResource, self).__init__()

    async def render_get(self, request):
        payload =
datetime.datetime.now().strftime("%Y-%m-%d %H:%M:%S").encode('ascii')
        return aiocoap.Message(code=aiocoap.Code.CONTENT, payload=payload)

class BlockResource(resource.Resource):
    def __init__(self):
        super(BlockResource, self).__init__()
        self.content = ("123456789ABCDEF\n" * 16).encode("ascii")

    async def render_get(self, request):
        return aiocoap.Message(code=aiocoap.Code.CONTENT, payload=self.content)

logging.basicConfig(level=logging.INFO)
logging.getLogger("coap-server").setLevel(logging.DEBUG)

def main():
    root = resource.Site()
    root.add_resource(('.well-known', 'core'),
resource.WKCResource(root.get_resources_as_linkheader))
    root.add_resource(('time',), TimeResource())
    root.add_resource(('other', 'block'), BlockResource())
    asyncio.Task(aiocoap.Context.create_server_context(root))
    asyncio.get_event_loop().run_forever()

if __name__ == "__main__":
main()
```

（1）增加 BlockResource 资源

在 test-server.py 中增加一个 BlockResource 类，该类与 TimeResource 相同也继承自 resource.Resource 类。

（2）定义响应内容

在 BlockResource 类的构造函数中定义一个 self.content 成员，在 GET 处理函数 render_get 中把 self.content 作为响应负载返回至 CoAP 客户端。self.content 由 16 组相同的字符串“123456789ABCDEF\n”组成，单组字符串长度为 16，self.content 的总长为 256 字节。若 CoAP 客户端设置的响应分包大小为 32，共需要 8 次分块传输；若 CoAP 客户端设置的响应分包大小为 64，共需要 4 次分包传输。

```
class BlockResource(resource.Resource):
    def __init__(self):
        super(BlockResource, self).__init__()
        self.content = ("123456789ABCDEF\n" * 16).encode("ascii")
    async def render_get(self, request):
        return aiocoap.Message(code=aiocoap.Code.CONTENT, payload=self.content)
```

2. 动手测试

（1）运行 CoAP 服务器

修改 test-server.py 之后，在树莓派 3 代中新建一个控制台并在控制台中重新运行 test-server.py。

```
python3 test-server.py
```

（2）运行 CoAP 客户端

此处使用 libcoap 中的 coap-client 工具作为 CoAP 客户端，在 Linux PC 控制台中输入：

```
coap-client -m get -b 64 coap://192.168.0.6/other/block
```

- -m get 表示 CoAP 请求方法为 GET 方法。
- -b 64 表示 CoAP 请求中包含块传输选项，分包大小为 64。

若 CoAP 服务器正确处理客户端请求，Linux 控制台将会获得以下内容：

```
v:1 t:CON c:GET i:17ed {} [ ]
123456789ABCDEF
123456789ABCDEF
123456789ABCDEF
123456789ABCDEF
v:1 t:CON c:GET i:17ee {} [ ]
123456789ABCDEF
123456789ABCDEF
123456789ABCDEF
123456789ABCDEF
v:1 t:CON c:GET i:17ef {} [ ]
123456789ABCDEF
123456789ABCDEF
123456789ABCDEF
123456789ABCDEF
v:1 t:CON c:GET i:17f0 {} [ ]
123456789ABCDEF
123456789ABCDEF
123456789ABCDEF
123456789ABCDEF
```

从控制台的输出内容可以发现，CoAP 服务器共返回 256 字节内容，分 4 次传输至客户端，每组大小为 64 字节。通过调整 -b 参数可修改分块传输大小，例如当设置“ -b 32”时，CoAP 服务器将把响应分成 8 组并依次传输至客户端。

7.3.4 aiocoap 树莓派 GPIO 示例

Python 作为一款通用的计算机编程语言具有很好的硬件操作能力。作为树莓派中的默认编程语言，使用 Python 可操作树莓派的 GPIO、SPI、UART 和 I2C 等外设，通过这些外设不但可以控制执行机构，也可以获取传感器检测结果。本小节以 GPIO 为例，说明如何把 CoAP 与 GPIO 控制结合在一起。

在 aiocoap 树莓派 GPIO 示例中，树莓派 3 代依然作为 CoAP 服务器而另一台 Linux 主机作为 CoAP 客户端。树莓派中使用 RPi.GPIO 扩展库控制 LED，该 LED 与树莓派扩展插座的第 1 脚相连，高电平可打开 LED，低电平可熄灭 LED。CoAP 客户端通过 JSON 类型负载控制 LED 点亮或熄灭，JSON 负载包含一组 JSON 对象，对象的键名为“value”，键值为整数类型的 0 或 1，0 表示 LED 熄灭而 1 表示 LED 点亮。aiocoap 树莓派 GPIO 示例如图 7-4 所示。

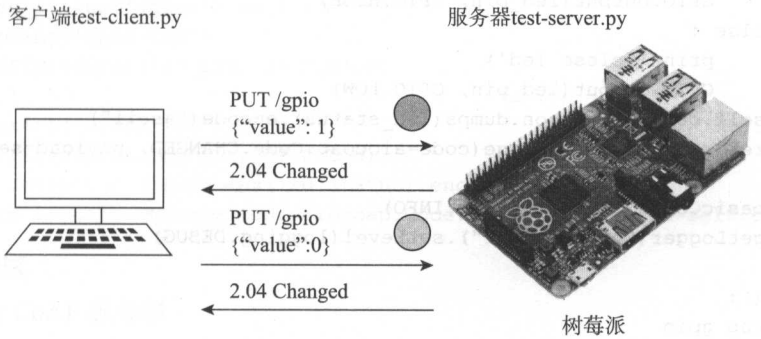


图 7-4　aiocoap 树莓派 GPIO 示例

1. 准备工作

一般情况下，树莓派 3 代中已经默认安装了 RPi.GPIO，如果需要把 RPi.GPIO 升级到最新版本，可在树莓派控制台中输入以下指令：

```
# 升级RPi.GPIO扩展库
sudo pip3 install RPi.GPIO --upgrade
```

2. 服务器实现

代码清单7-4　rpi_gpio_server.py

```
#!/usr/bin/env python3
import logging

import asyncio
import aiocoap.resource as resource
import aiocoap

import RPi.GPIO as GPIO
import json

led_pin = 11
class GPIOResource(resource.Resource):
    def __init__(self):
        super(GPIOResource, self).__init__()
        led_status = {'value': 0}
        self.content = json.dumps(led_status).encode("ascii")
```

```
    async def render_get(self, request):
        return aiocoap.Message(code=aiocoap.Code.CONTENT, payload=self.content)

    async def render_put(self, request):
        print('PUT payload: %s' % request.payload)
        led_status = json.loads(request.payload.decode())
        if led_status['value'] == 1 :
            print('open led')
            GPIO.output(led_pin, GPIO.HIGH)
        else :
            print('close led')
            GPIO.output(led_pin, GPIO.LOW)
        self.content = json.dumps(led_status).encode("ascii")
        return aiocoap.Message(code=aiocoap.Code.CHANGED, payload=self.content)

logging.basicConfig(level=logging.INFO)
logging.getLogger("coap-server").setLevel(logging.DEBUG)

def main():
    # setup gpio
    GPIO.setmode(GPIO.BOARD)
    GPIO.setup(led_pin, GPIO.OUT)

    # Resource tree creation
    root = resource.Site()
    root.add_resource(('.well-known', 'core'),
resource.WKCResource(root.get_resources_as_linkheader))
    root.add_resource(('gpio',), GPIOResource())
    asyncio.Task(aiocoap.Context.create_server_context(root))
    asyncio.get_event_loop().run_forever()

if __name__ == "__main__":
    main()
```

（1）设置 GPIO 为输出状态

通过 GPIO.setmode（GPIO.BOARD）设置树莓派 GPIO 编号方式，并通过 GPIO.setup 把第 11 引脚设置为输出状态。

```
import RPi.GPIO as GPIO

led_pin = 11
def main():
    # setup gpio
    GPIO.setmode(GPIO.BOARD)
    GPIO.setup(led_pin, GPIO.OUT)
# 省略部分代码
```

（2）处理 PUT 请求

CoAP 请求负载为 JSON 格式，例如 {“value”：1}。PUT 请求处理函数中 request.

payload 为 bytes 类型，可通过 decode 函数把 bytes 类型转换为字符串类型，再通过 json.loads 函数转化为 Python 字典类型，Python 字典类型和 JSON 类型存在直接对应关系。led_status 对应 LED 具体状态，led_status['value'] 对一个 LED 打开或关闭状态，若 led_status['value'] 等于 1，则打开 LED。

```
async def render_put(self, request):
    print('PUT payload: %s' % request.payload)
    led_status = json.loads(request.payload.decode())
    if led_status['value'] == 1 :
        print('open led')
        GPIO.output(led_pin, GPIO.HIGH)
    else :
        print('close led')
        GPIO.output(led_pin, GPIO.LOW)
    self.content = json.dumps(led_status).encode("ascii")
    return aiocoap.Message(code=aiocoap.Code.CHANGED, payload=self.content)
```

3. 动手测试

（1）运行 CoAP 服务器

在树莓派 3 代中新建一个控制台，在控制台中输入：

```
python3 rpi_gpio_server.py
```

（2）运行 CoAP 客户端

在 Linux PC 主机中通过 coap-client 工具点亮 LED，可输入以下指令点亮 LED：

```
coap-client -m put -e {\"value\":1} coap://192.168.0.6/gpio
```

- -m put 表示 CoAP 请求方法为 PUT 方法。
- -e {\"value\":1} 表示 CoAP 请求负载为字符串形式的 {\"value\":1}。

若熄灭 LED，可以把请求负载中的“1”改为“0”。

```
coap-client -m put -e {\"value\":0} coap://192.168.0.6/gpio
```

7.4　node-coap

Node.js 是最近非常活跃的脚本语言，Node.js 一般用于 Web 开发，但是随着 Node.js 本身技术的发展和开源社区的努力，Node.js 也逐渐进入了嵌入式开发领域。Node.js 是一个基于 Chrome V8 引擎的 JavaScript 运行环境。Node.js 使用了事件驱动、非阻塞式 I/O 的模型，Node.js 既轻量又高效，是新一代计算机开发语言。与前面介绍的 Python 3 相似，Node.js 也有自己的包管理工具——npm。

node-coap [⊖] 便是使用 Node.js 语言实现 CoAP 的开源框架，相比于语法略显沉重的

⊖ https://github.com/mcollina/node-coap。

Java，Node.js 更易于学习，只编写少量的代码便可实现各种各样不同功能的 CoAP 服务器或客户端。与 Python 相似，Node.js 非常适合 CoAP 初学者。

7.4.1 Node.js 安装

由于 Node.js 并不是树莓派的默认安装软件，所以开始 node-coap 入门示例之前需在树莓派中正确安装 Node.js。

Node.js 可以在多种平台运行，Node.js 提供了各种不同平台的安装包和源代码包。本节将介绍如何在树莓派 3 代中安装 Node.js，树莓派 3 代中安装 Node.js 的方法与其他 Linux 主机中安装 Node.js 的方法非常相似，可以分为软链接方式安装、代码仓库安装和源代码安装等，本节将详细介绍这三种方法。而 Windows 平台的安装方法非常简单，本节将不再详细说明。由于 Node.js 的版本众多，本节以 4.7.0 版本为例说明 Node.js 安装过程。

1. 软链接方式安装 Node.js

下面介绍如何在树莓派 3 中使用软链接方式安装 Node.js。Node.js 提供了 ARM 平台安装包文件，这些安装包可分为三种不同的 ARM 指令集——ARMv6、ARMv7 和 ARMv8。树莓派 1 代 B+ 需选择 ARMv6 版本安装包，更高硬件版本的树莓派需选择 ARMv7 版本安装包。若不清楚树莓派的 ARM 指令集版本也没有关系，可查看 proc/cpuinfo 文件获得更多信息的信息，如图 7-5 所示为树莓派 1 代 B+ 的 CPU 版本信息，而如图 7-6 所示为树莓派 3 代的 CPU 版本信息。

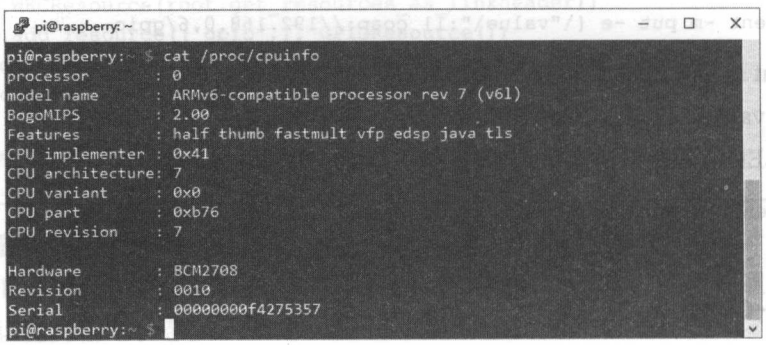


图 7-5　树莓派 1 代 B+ CPU 信息

在树莓派 3 中通过软链接方式安装 Node.js，具体步骤如下：

```
# 新建一个文件夹用于存放Node.js安装文件
mkdir -p software
# 进入software目录
cd software
# 获取Node.js安装包 指定ARM指令集为 armv7
wget https://nodejs.org/dist/v4.7.0/node-v4.7.0-linux-armv7l.tar.xz
# 解压Node.js安装包
```

```
xz -d node-v4.7.0-linux-armv7l.tar.xz
tar -xf node-v4.7.0-linux-armv7l.tar
# 为node和npm创建软链接
sudo ln -s ~/software/node-v4.7.0-linux-armv7l/bin/node /usr/local/bin/node
sudo ln -s ~/software/node-v4.7.0-linux-armv7l/bin/npm /usr/local/bin/npm
```

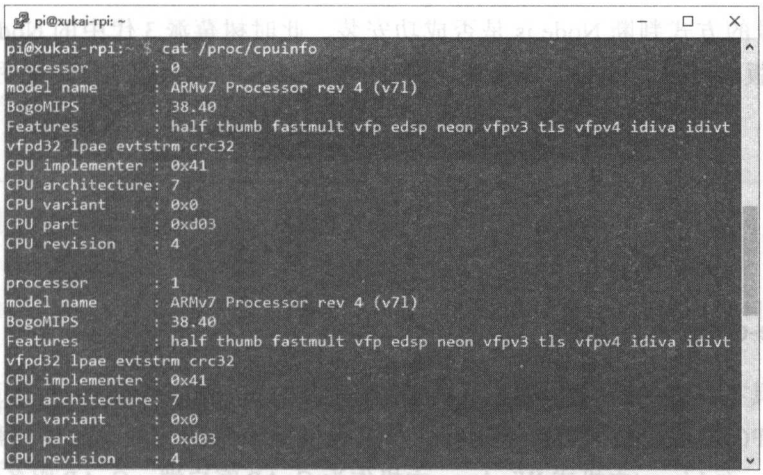


图 7-6　树莓派 3 代 CPU 信息

2. 代码仓库方式安装 Node.js

除了软链接方式安装 Node.js 之外，也可使用代码仓库方式安装 Node.js，本书推荐使用代码仓库方式安装 Node.js。与软链接方式不同，安装过程中并不需要指定 Node.js 的具体版本编号，代码仓库方式总是以最新版本的 Node.js 为准。若在树莓派 3 代中安装 Node.js 可参考以下指令：

```
curl -sL https://deb.nodesource.com/setup_4.x | sudo -E bash -
sudo apt-get install -y nodejs
```

3. 源代码方式安装 Node.js

相对于以上两种方法，源代码方式安装 Node.js 需要消耗较多时间，具体安装过程如下：

```
# 新建一个文件夹用于存放Node.js安装文件
mkdir -p software
# 进入software目录
cd software
wget https://nodejs.org/dist/v4.7.0/node-v4.7.0.tar.gz
# 解压Node.js源代码包
tar -xzvf node-v4.7.0.tar.gz
# 进入Node.js源代码文件夹
cd node-v4.7.0
# 生成配置文件
./configure
# 编译源文件，编译过程时间较长请耐心等待
```

```
make
# 执行安装过程
sudo make install
```

4. 检查安装结果

在树莓派 3 中新建一个控制台，在控制台中依次输入“node -v”和“npm -v”，通过检查版本号的方式判断 Node.js 是否成功安装，此时树莓派 3 代中的 Node.js 版本号为 v4.7.0，npm 版本号为 2.15.11。如果从控制台中观察到正确的版本信息，说明 Node.js 已经安装成功。

```
node -v
v4.7.0
npm -v
2.15.11
```

7.4.2 node-coap 入门示例

Node.js 具有良好的跨平台特性，入门示例涉及的 client.js 和 server.js 均可在 Windows 主机、Linux PC 主机或是树莓派上运行。与其他的几个示例相似，此处树莓派依然作为 CoAP 服务器，而 Linux 主机或 Windows 主机作为 CoAP 客户端。CoAP 服务器中具有一个系统时间服务（资源），通过该服务（资源）可获取树莓派的系统时间。Node-coap 入门示例的大致工作流程如图 7-7 所示。

node-coap 项目的更多内容可参考：https://github.com/mcollina/node-coap。

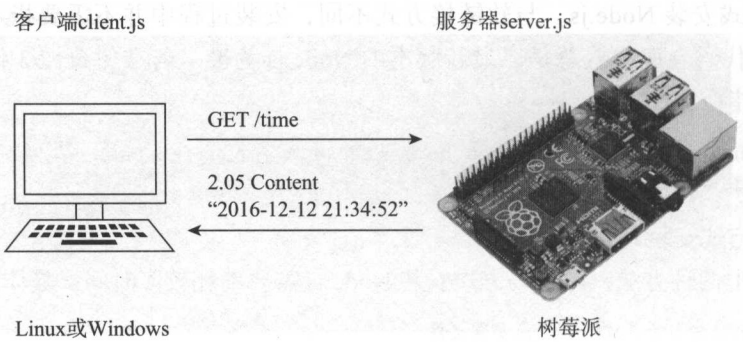


图 7-7　Node.js 入门示例

1. 服务器实现

server.js 实现代码如下：

代码清单7-5　server.js

```
const coap = require('coap')
const server = coap.createServer()
```

```
server.on('request', function(req, res) {
    console.log(req.url)
    console.log(req.method)
    if (req.method == 'GET' && req.url.split('/')[1] == 'time') {
        res.end(new Date().toISOString())
    }
})

server.listen(function() {
  console.log('server started')
})
```

server.js 中包含一个 time 路由，且该路由仅支持 GET 方法。

（1）创建 CoAP 服务并侦听请求

通过 createServer 方法构造一个 CoAP Server，并通过 listen 方法侦听来自 5683 端口的 CoAP 请求。

```
const coap = require('coap')
const server = coap.createServer()
server.listen(function() {
  console.log('server started')
})
```

（2）处理 CoAP 请求

若 CoAP 服务器接收到来自 CoAP 客户端的请求，将会触发 request 消息，在该消息回调函数中对请求内容变量 req 进行处理。req.method == 'GET' && req.url.split('/')[1] == 'time' 表示该 CoAP 仅处理路由为“time”的请求，且方法必须为 CoAP GET，其他方法或路由该 CoAP 服务器将不作响应。

```
server.on('request', function(req, res) {
    console.log(req.url)
    console.log(req.method)
    if (req.method == 'GET' && req.url.split('/')[1] == 'time') {
        res.end(new Date().toISOString())
    }
})
```

（3）返回系统时间

在处理请求过程时，CoAP 服务器通过 res.end(new Date().toISOString()) 将树莓派的系统时间反馈至客户端。在 node-coap 框架中 response.end() 和 response.write() 均可向 CoAP 响应中填充内容，response.end() 包含填充与执行两个动作，而 response.write() 仅包括填充一个动作。换句话说，response.write() 填充 CoAP 响应之后还应调用一次 response.end()。

2. 客户端实现

client.js 实现代码如下：

代码清单7-6 client.js

```
const coap  = require('coap')
const req    = coap.request({host:'<raspberry_ip_address>', pathname:'time',
method:'GET'})

req.on('response', function(res) {
    console.log('response code: ' + res.code)
    console.log('response payload: ' + res.payload)})
}
```

（1）创建 CoAP 请求

使用 request 方法构造 CoAP 请求，在 request 参数中写入一个 JavaScript 对象，其中 host 表示 CoAP 服务器 IP 地址，此处需使用树莓派的实际 IPv4 地址；pathname 表示路由名称，此处为“time”；method 表 CoAP 请求方法，此时请求方法为“GET”。

```
coap.request({host:'<raspberry_ip_address>', pathname:'time', method:'GET'})
```

（2）响应结果输出至控制台

若客户端接收到来自 CoAP 服务器的正确响应，那么就会触发 response 事件，res 变量中保存 CoAP 响应的全部内容，入门示例代码把响应码 res.code 和响应负载 res.payload 均输出至控制台中。

```
req.on('response', function(res) {
    console.log('response code: ' + res.code)
    console.log('response payload: ' + res.payload)
})
```

3. 动手测试

（1）运行 CoAP 服务器

运行 CoAP 服务器之前需要在 server.js 同级目录中先使用 npm 工具安装 node-coap，npm 是一种 Node.js 的包管理工具，通过 npm install node-coap 可完成 node-coap 的安装动作，安装完成之后，将在 server.js 同级目录中增加一个名为 node_modules 目录。在树莓派控制台中输入以下指令以启动 CoAP 服务器：

```
# 通过npm工具安装node-coap
npm install node-coap
# 在树莓派3中启动CoAP服务器
node server.js
# 控制台输出
server started
```

（2）启动 CoAP 客户端

在 Windows 或者 Linux PC 主机中输入以下指令以运行 CoAP 客户端：

```
# 通过npm安装node-coap
npm install node-coap
# 运行coap客户端
node client.js
# 控制台输出
response code: 2.05
response payload: 2016-08-01T08:14:32.060Z
```

此时 CoAP 响应码为 2.05，说明 CoAP 服务器成功处理了 CoAP 请求，CoAP 响应负载为树莓派的系统时间。

7.4.3　node-coap 媒体类型示例

node-coap 入门示例中 server.js 仅返回文本类型的响应负载，无法根据 CoAP 客户端的期望返回不同媒体类型的响应负载。CoAP 规定，CoAP 请求中可使用 Accept 选项设置期望媒体类型，而 CoAP 响应中可使用 Content-Format 设置负载媒体类型。在本小节的示例中，json_server.js 将通过 CoAP 请求中的 Accept 选项返回不同的媒体类型，json_server.js 的工作流程如图 7-8 所示。

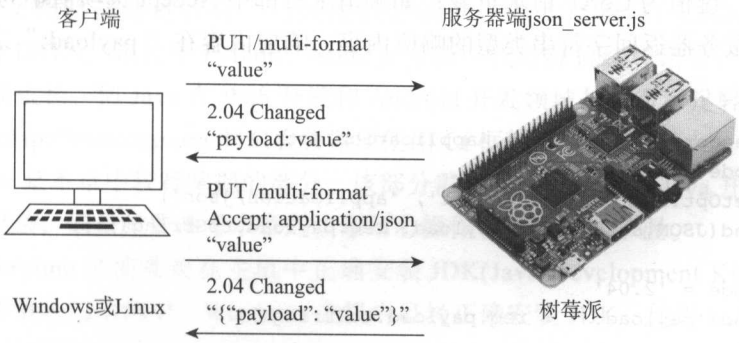


图 7-8　可返回 JSON 响应示例

1. 服务器实现

json_server.js 具体代码如下：

代码清单7-7　json_server.js

```
const coap = require('coap')
const server = coap.createServer()

server.on('request', function(req, res) {
    console.log('method:' + req.method + ' url:' + req.url)
    if (req.method == 'PUT' && req.url.split('/')[1] == 'multi-format') {
        if (req.headers['Accept'] == 'application/json') {
```

```
            res.code = '2.04'
            res.setOption('Content-Format', 'application/json')
            res.end(JSON.stringify({payload: req.payload.toString()}))
        } else {
            res.code = '2.04'
            res.end('payload: ' + req.payload.toString())
        }
    }
})

server.listen(function() {
    console.log('server started')
})
```

（1）处理 CoAP 请求

CoAP 服务器仅处理 multi-format 路由，且 CoAP 请求的方法为 PUT。

```
req.method == 'PUT' && req.url.split('/')[1] == 'multi-format'
```

（2）根据请求首部选项返回内容

如果请求首部中包含 Accept 选项，且 Accept 选项值为“application/json”，那么 CoAP 服务器返回 JSON 类型的响应内容，响应内容包括一个 JSON 对象，该 JSON 对象中的键名为“payload”，键值为 CoAP 请求负载；如果请求首部中 Accept 选项值不为“application/json”，CoAP 服务器返回字符串类型的响应内容，响应内容在“payload:”之后增加 CoAP 请求中的负载。

```
if (req.headers['Accept'] == 'application/json') {
    res.code = '2.04'
    res.setOption('Content-Format', 'application/json')
    res.end(JSON.stringify({payload: req.payload.toString()}))
} else {
    res.code = '2.04'
    res.end('payload: ' + req.payload.toString())
}
```

2. 动手测试

（1）运行 CoAP 服务器

在树莓派中新建控制台，在控制台中输入以下内容：

```
# 运行coap服务器
node json_server.js
```

（2）运行 CoAP 客户端

此处使用 coap-client 工具作为 CoAP 客户端，在 Linux PC 控制台中输入以下内容：

```
coap-client -m put -A 50 -e value coap://192.168.0.6/multi-format
```

- -m put 表示 CoAP 请求方法为 PUT 方法。

- -A 50 表示 CoAP 请求中包含 Accept 选项且选项值为 50，CoAP 媒体类型部分规定 50 即代表 application/json。
- -e value 表示 CoAP 请求负载为“value”。

若 CoAP 服务器正确处理客户端请求，CoAP 服务器将返回一个 JSON 类型响应。

```
v:1 t:CON c:PUT i:e7d9 {} [ ]
{"payload":"value"}
```

若去除 coap-client 命令参数中的“-A 50”：

```
coap-client -m put -e value coap://192.168.0.6/multi-format
```

那么 CoAP 服务器将返回字符串形式的响应负载“payload: value”。

```
v:1 t:CON c:PUT i:42b0 {} [ ]
payload: value
```

7.5 Californium

Californium[⊖]是 Eclipse IoT 项目的一部分，Californium 可简称为 Cf。Californium 是使用 Java 语言实现的 CoAP 开源框架。Californium 包含 CoAP 服务器和 CoAP 客户端方面绝大多数 RFC 文档所描述的特性。与其他轻量级的开源实现不同，Californium 还包括 CoAP 安全部分 DTLS。相比于 Python、Node.js 等新一代计算机编程语言，Java 语法较为沉重、学习周期也长，但 Java 在 Web 开发和 Android 开发领域却占有举足轻重的地位。本书前文提及的 coap://wsncoap.org 测试服务器便使用 Cf 框架开发。

Californium 是本章中较难掌握的部分，该部分需要用户已经具备 Java 开发经验。虽然这些部分看似复杂，但是只要参考本章的步骤耐心操作也可以正确掌握。

使用 Californium 之前需要在主机中正确安装 JDK(Java Development Kit) 和 Java 集成开发工具。在本节入门示例中，Windows 主机内已经正确安装 JDK，使用 Eclipse 作为 Java 集成开发工具。

7.5.1 准备工作

与之前入门示例相似，树莓派 3 代依然作为 CoAP 服务器，而另一台 Windows 或 Linux 主机作为 CoAP 客户端。为了保证入门示例中生成的可执行 jar 文件可在树莓派或 Windows 主机中顺利运行，需在树莓派和 Windows 主机中正确安装 JDK。由于树莓派 3 中已经默认安装了 JDK，而多数 Windows 主机可参考以下步骤完成 JDK 的安装。

1. Windows 主机下 JDK 安装

（1）获取 JDK 安装文件

⊖ https://www.eclipse.org/californium/。

前往 Oracle 官网下载最新版本的 JDK 安装文件，并安装到 Windows 主机中，本节 JDK 的安装目录为“D:\Program Files\Java\jdk1.8.0_111”。

（2）进入环境变量修改界面

为了正常使用 JDK，需在 Windows 中设置环境变量。在桌面右击“我的电脑”选择属性，进入“系统属性”界面，选择“高级”选项卡再点击“环境变量”。如图 7-9 所示。

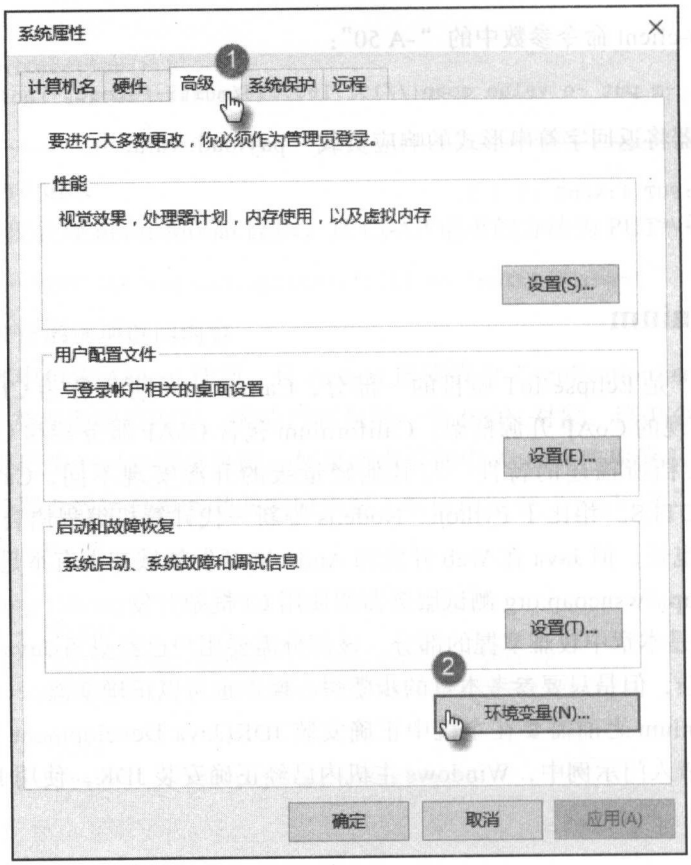


图 7-9　系统属性界面

（3）增加 JAVA_HOME 变量

在系统变量中增加 JAVA_HOME 变量，在变量值中写入 JDK 具体安装路径，例如 JDK 的安装路径为“D:\Program Files\Java\jdk1.8.0_131”。如图 7-10 所示。

（4）增加 CLASSPATH 变量

在系统变量中再增加一个 CLASSPATH 变量，在变量值中增加 lib 目录和 tools.jar 文件，此处的变量值为“.;%JAVA_HOME%\lib;%JAVA_HOME%\lib\tools.jar”。如图 7-11 所示。

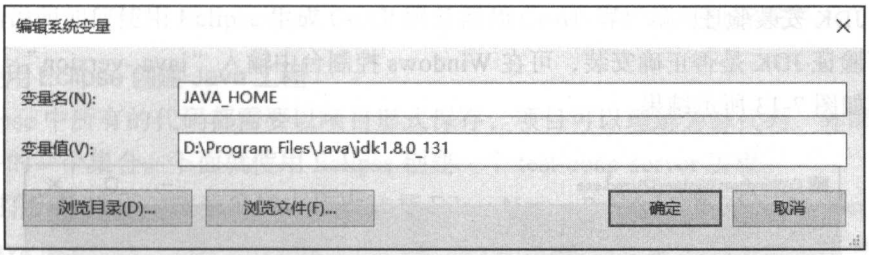


图 7-10　增加 JAVA_HOME 系统变量

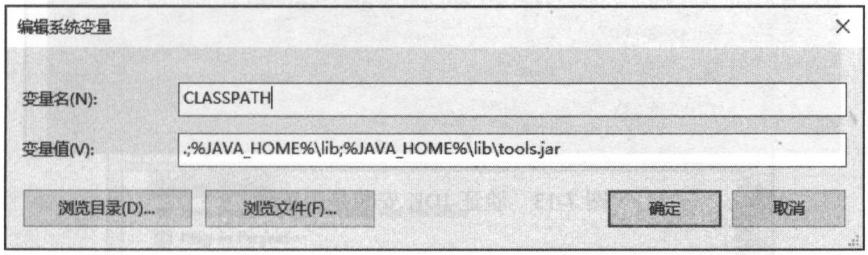


图 7-11　增加 CLASSPATH 系统变量

（5）修改 PATH 变量

最后修改 PATH 变量。在 PATH 变量中通过“新建”方法增加两项，一项为“%JAVA_HOME%\bin”，另一项为“%JAVA_HOME%\jre\bin”，如图 7-12 所示。

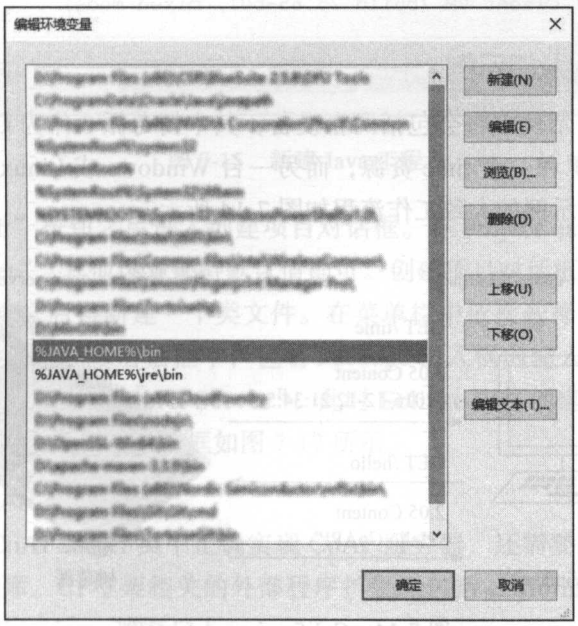


图 7-12　修改 PATH 变量

（6）JDK 安装验证

为了验证 JDK 是否正确安装，可在 Windows 控制台中输入“java -version”，若安装正确可获得如图 7-13 所示结果。

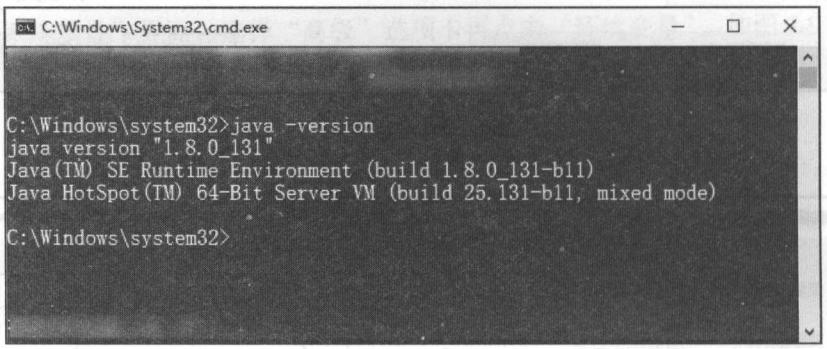


图 7-13　验证 JDK 安装是否正确

2. 树莓派中验证 JDK 安装

对于树莓派 3 代来说 JDK 已经默认安装软件，在树莓派 3 控制台中运行“java -version”也可以查询此时树莓派 3 中的 JDK 版本编号。树莓派 3 代控制台的输出结果如下：

```
java -version
java version "1.8.0_65"
Java(TM) SE Runtime Environment (build 1.8.0_65-b17)
Java HotSpot(TM) Client VM (build 25.65-b01, mixed mode)
```

7.5.2　Californium 入门示例

Californium 入门示例中将会包括两台设备，其中树莓派 3 作为 CoAP 服务器，该服务器提供一个 hello 资源和一个 time 资源，而另一台 Windows 或 Linux 主机作为 CoAP 客户端。Californium 入门示例的大致工作流程如图 7-14 所示。

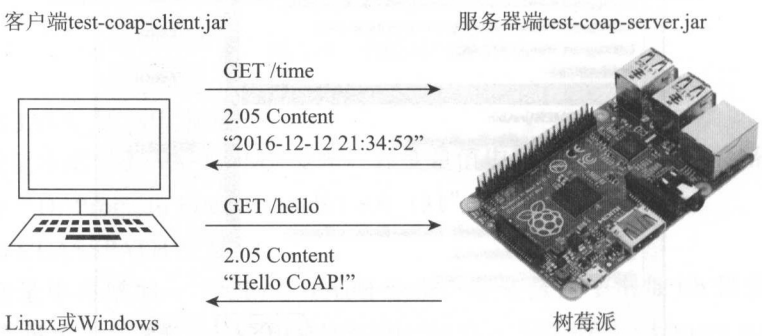


图 7-14　Californium 入门示例

下面说明如何使用 Eclipse 生成 CoAP 服务器和 CoAP 客户端可执行程序。

1. 使用 Eclipse 创建 Java 工程

Eclipse 中所有的代码都需要以项目形式保存，项目可以理解为源代码、外部程序库和配置文件的一个集合。下面就使用 Eclipse 创建一个 test-coap-server 工程。

1）打开 Eclipse，在菜单栏中依次选择 File→New→Project，在 New Project 对话框中选择“Java Project”，单击 Next 按钮进入下一步。New Project 对话框如图 7-15 所示。

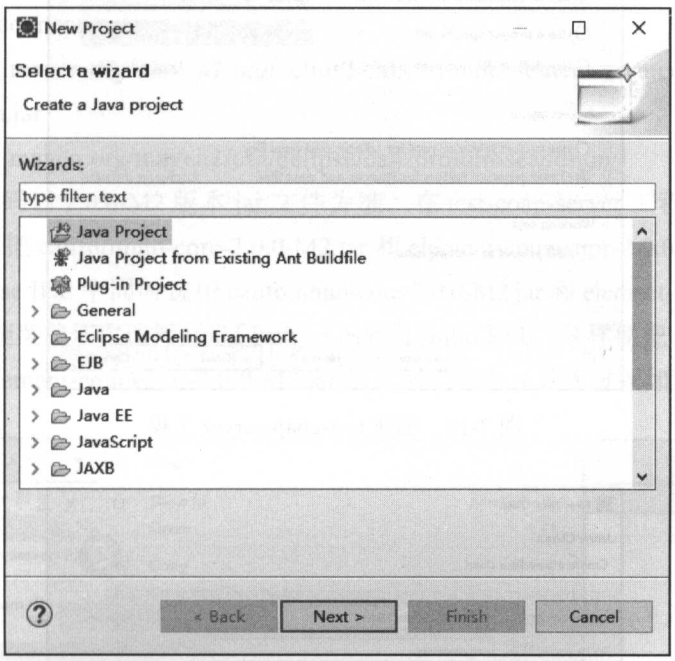


图 7-15　新建 Java 工程

2）单击“Finish”按钮之后进入创建项目对话框。在 Project name 输入框中输入项目名称“test-coap-server”，其他设置保持默认值即可。创建项目对话框如图 7-16 所示。

3）完成工程创建之后再新建一个类文件。在菜单栏中依次选择“File→New→Class”进入新建类向导对话框。在该对话框中，包名 Package 输入框内输入“org.wsncoap”；类名 Name 输入框内输入“HelloCoAPServer”，此时 Eclipse 中将创建一个名为 HelloCoAP-Server.java 的文件。新建类向导对话框如图 7-17 所示。

2. 获取 Cf 相关 JAR 文件

若需要在 HelloCoAPServer 类中正确实现 CoAP 服务器，还需要在工程中引入 Cf 框架相关的外部程序扩展库。Cf 框架相关的外部程序扩展库包括 californium-core.jar、element-connector.jar 和 scandium.jar。其中：

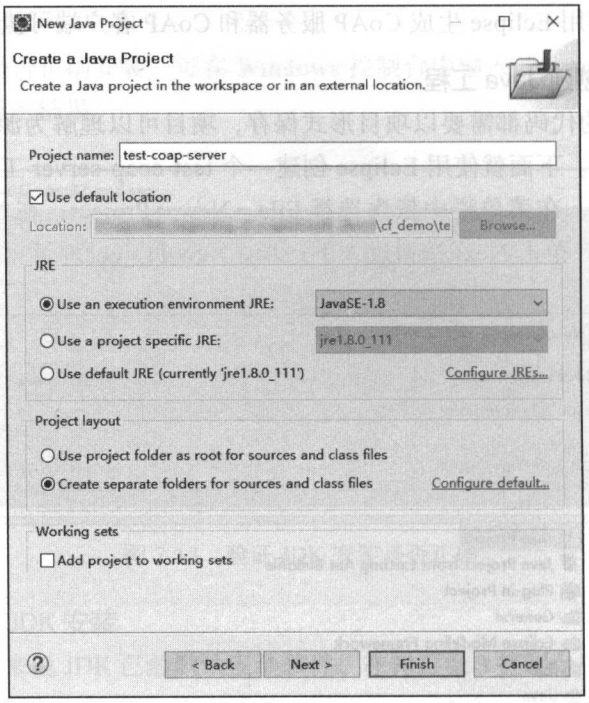


图 7-16　创建 test-coap-server 工程

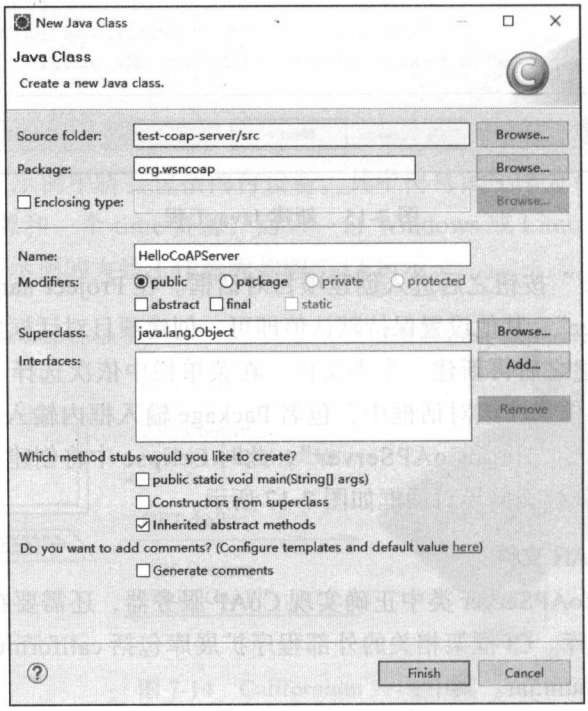


图 7-17　新建 HelloCoAPServer 类

1）californium-core.jar 包括 CoAP 核心部分。

2）element-connector.jar 包括适用于 UDP 和 DTLS 的 Java 套接字抽象层。

3）scandium.jar 包括 DTLS 部分功能。

相关 Jar 文件可前往 Maven 中央仓库获取，californium-core.jar、element-connector.jar 和 scandium.jar 的 maven 仓库网址如下：

1）californium-core.jar：

http://central.maven.org/maven2/org/eclipse/californium/californium-core/

2）element-connector.jar：

http://central.maven.org/maven2/org/eclipse/californium/element-connector/

3）scandium.jar：

http://central.maven.org/maven2/org/eclipse/californium/scandium/

本节示例代码以 2.0.0-M2 版本 jar 文件为准，在 test-coap-server 工程中新建一个名为 lib 的文件夹，并把 californium-core-2.0.0-M2.jar 和 element-connector-2.0.0-M2.jar 存入 lib 文件夹中。在 Eclipse IDE 中同时选中 californium-core-2.0.0-M2.jar 和 element-connector-2.0.0-M2.jar，右击进入快捷菜单依次选择 Build Path → Add to Build Path。这样便把 element- connector-2.0.0-M2.jar 和 element-connector-2.0.0-M2.jar 加入到工程中，具体过程如图 7-18 所示。

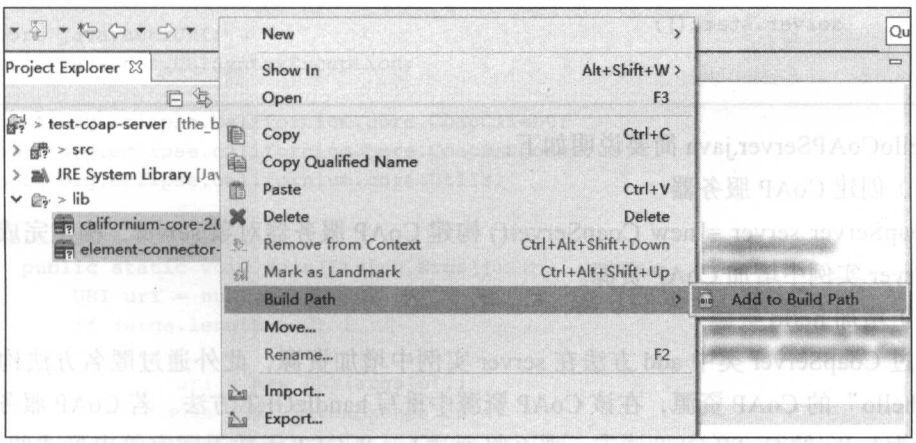


图 7-18　增加外部程序扩展库

3. HelloCoAPServer.java

在 HelloCoAPServer.java 中增加一个 main 函数，在 main 函数中增加两个 CoAP 资源：一个名为“hello”，另一个名为“time”。HelloCoAPServer.java 的具体内容如下：

代码清单7-8　HelloCoAPServer.java

```
package org.wsncoap;
```

```
import java.text.SimpleDateFormat;
import java.util.Date;

import org.eclipse.californium.core.CoapResource;
import org.eclipse.californium.core.CoapServer;
import org.eclipse.californium.core.coap.CoAP.ResponseCode;
import org.eclipse.californium.core.server.resources.CoapExchange;

public class HelloCoAPServer {
    public static void main(String[] args) {
        CoapServer server = new CoapServer();
        server.add(new CoapResource("hello") {
            @Override
            public void handleGET(CoapExchange exchange) {
                exchange.respond(ResponseCode.CONTENT, "Hello CoAP!");
            }
        });
        server.add(new CoapResource("time") {
            @Override
            public void handleGET(CoapExchange exchange) {
                Date date = new Date();
                exchange.respond(ResponseCode.CONTENT,
                    new SimpleDateFormat("yyyy-MM-dd HH:mm:ss").format(date));
            }
        });
        server.start();
    }
}
```

HelloCoAPServer.java 简要说明如下：

（1）创建 CoAP 服务器

CoapServer server = new CoapServer() 构建 CoAP 服务器对象 server，构建完成之后便可向 server 实例中增加 CoAP 资源。

（2）增加 hello 资源

通过 CoapServer 类中 add 方法在 server 实例中增加资源，此外通过匿名方法构造一个名为“hello”的 CoAP 资源，在该 CoAP 资源中重写 handleGET 方法。若 CoAP 服务器接收到 URI 为 hello 的 CoAP GET 请求，那么将通过 handleGET 函数返回字符串形式的“Hello CoAP!”。

```
server.add(new CoapResource("hello") {
    @Override
    public void handleGET(CoapExchange exchange) {
        exchange.respond(ResponseCode.CONTENT, "Hello CoAP!");
    }
});
```

（3）增加 time 资源

CoAP 服务器除了 hello 资源之外还包括 time 资源。与 hello 资源不同，time 资源返回“年 – 月 – 日 时 : 分 : 秒”这样固定格式的字符串内容。

```
server.add(new CoapResource("time") {
    @Override
    public void handleGET(CoapExchange exchange) {
        Date date = new Date();
        exchange.respond(ResponseCode.CONTENT,
            new SimpleDateFormat("yyyy-MM-dd HH:mm:ss").format(date));
    }
});
```

（4）启动 CoAP 服务器

最后使用 start 方法启动 CoAP 服务器。

4. GETClient.java

完成 CoAP 服务器代码编写之后我们再进行 CoAP 客户端代码的编写。在 Eclipse 中新建一个名为 test-coap-client 工程，通过类创建向导新建一个名为 GETClient.java 的源文件，GETClient.java 的具体代码如下。

代码清单7-9　GETClient.java

```
package org.wsncoap;

import java.net.URI;
import java.net.URISyntaxException;

import org.eclipse.californium.core.CoapClient;
import org.eclipse.californium.core.CoapResponse;
import org.eclipse.californium.core.Utils;

public class GETClient {
    public static void main(String args[]) {
        URI uri = null;
        if (args.length > 0) {
            try {
                uri = new URI(args[0]);
            } catch (URISyntaxException e) {
                System.err.println("Invalid URI: " + e.getMessage());
                System.exit(-1);
            }

            CoapClient client = new CoapClient(uri);
            CoapResponse response = client.get();
            if (response!=null) {

                System.out.println(response.getCode());
                System.out.println(response.getOptions());
                System.out.println(response.getResponseText());
```

```
                System.out.println("\nADVANCED\n");
                System.out.println(Utils.prettyPrint(response));
            } else {
                System.out.println("No response received.");
            }
        }
    }
}
```

GETClient.java 简要说明如下：

（1）获取 CoAP URI

uri = new URI(args[0])：GETClient.java 通过命令行的第一个参数获取 CoAP URI。

（2）构造并发送 CoAP 请求

构造一个 CoAP 请求对象，通过 CoapClient 类中的 GET 方法发送 CoAP 请求。

```
CoapClient client = new CoapClient(uri);
CoapResponse response = client.get();
```

（3）获取响应并输出至控制台

若 CoAP 客户端接收到来自 CoAP 服务器的响应，那么 GETClient.java 将把响应码、响应选项和响应负载输出至控制台。其中：

- response.getCode() 可获取响应码。
- response.getOptions() 可获取响应选项。
- response.getResponseText() 可获取字符串形式的响应负载。

Cf 框架中也可以使用 Utils.prettyPrint 方法，把 response 变量格式化之后输出至控制台，通过 Utils.prettyPrint 方法格式化之后内容更容易被阅读。

```
System.out.println(response.getCode());
System.out.println(response.getOptions());
System.out.println(response.getResponseText());

System.out.println("\nADVANCED\n");
System.out.println(Utils.prettyPrint(response));
```

5. 动手测试

完成 CoAP 服务器和 CoAP 客户端的代码编写工作之后可进入动手测试环节。

（1）生成可执行 jar 文件

为了脱离 Eclipse 集成开发环境，可借助 Export 工具导出可运行 jar 文件，以 test-coap-server 工程为例说明如何生成可执行的 jar 文件。

1）选中 test-coap-server 工程，右击进入快捷菜单。

2）在快捷菜单中选择 Export，进入 Export 对话框。

3）在 Export 对话框中选择 Runnable JAR file，单击 Next 按钮进入下一步。Export 对话框如图 7-19 所示。

4）在 Runnable JAR file 对话框中选择 test-coap-server 工程，并选择合适的输出路径，最后单击 Finish 按钮完成所有的输出操作，该步骤操作过程如图 7-20 所示。

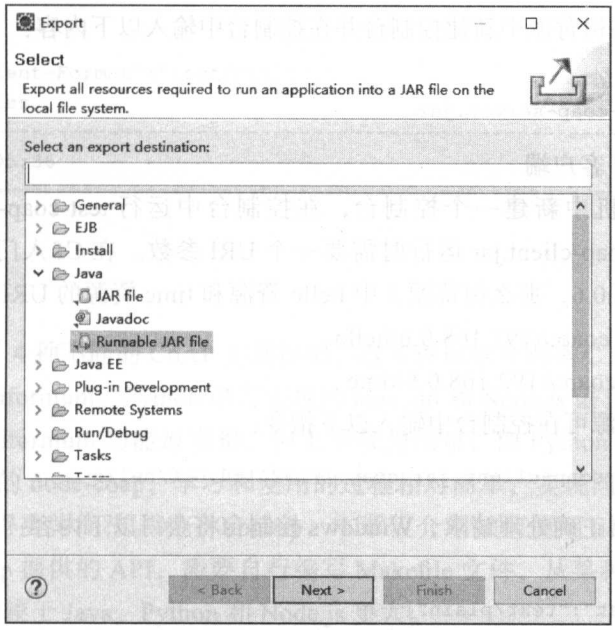


图 7-19　Export 对话框

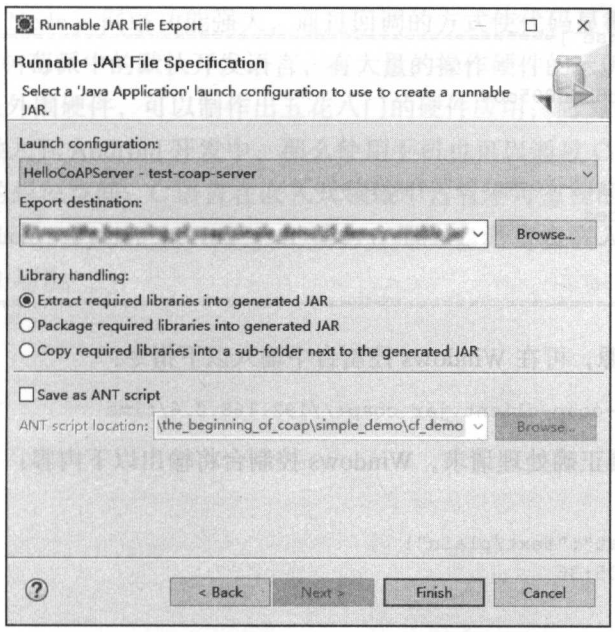


图 7-20　输出可执行 JAR 文件

（2）运行 CoAP 服务器

通过上面的步骤可以获得两个可执行 JAR 文件——test-coap-server.jar 和 test-coap-client.jar。为了在树莓派中运行 CoAP 服务器可使用 SFTP 工具把 test-coap-server.jar 传输至树莓派 3 代中。在树莓派中新建控制台并在控制台中输入以下内容：

```
# 运行CoAP服务器
java -jar test-coap-server.jar
```

（3）运行 CoAP 客户端

在 Windows 主机中新建一个控制台，在控制台中运行 test-coap-client.jar。与 CoAP 服务器不同，test-coap-client.jar 运行时需要一个 URI 参数。在 Cf 入门示例中树莓派 3 的 IPv4 地址为 192.168.0.6，那么树莓派 3 中 hello 资源和 time 资源的 URI 分别为：

- hello 资源：coap://192.168.0.6/hello。
- time 资源：coap://192.168.0.6/time。

若访问 hello 资源可在控制台中输入以下指令：

```
java -jar test-coap-client.jar coap://192.168.0.6/hello
```

若 CoAP 服务器正确处理请求，Windows 控制台将获得以下内容：

```
2.05
{"Content-Format":"text/plain"}
Hello CoAP!

ADVANCED

==[ CoAP Response ]=============================================
MID     : 32637
Token   : 75a4bb0bdc805d84
Type    : ACK
Status  : 2.05
Options: {"Content-Format":"text/plain"}
Payload: 11 Bytes
----------------------------------------------------------------
Hello CoAP!
================================================================
```

若访问 time 资源，可在 Windows 控制台中输入以下指令：

```
java -jar test-coap-client.jar coap://192.168.0.6/time
```

若 CoAP 服务器正确处理请求，Windows 控制台将输出以下内容：

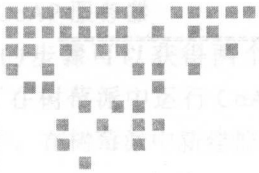
```
2.05
{"Content-Format":"text/plain"}
2016-12-19 20:05:36

ADVANCED
```

```
==[ CoAP Response ]=========================================
MID     : 30807
Token   : ae6ad1b628
Type    : ACK
Status  : 2.05
Options: {"Content-Format":"text/plain"}
Payload: 19 Bytes
------------------------------------------------------------
2016-12-19 20:05:36
============================================================
```

7.6　本章小结

本章总共介绍了 4 种不同的 CoAP 实现框架，这 4 种框架分别是 C 语言实现的 libcoap、Java 语言实现的 Californium、Python 语言实现的 aiocoap 和 Node.js 语言实现的 node-coap。Java 语言实现的 Californium 功能最完整，但上手较为困难；而 Python 语言实现的 aiocoap 和 Node.js 语言实现的 node-coap，学习和使用的过程相对简单，实现同样的功能 Python 和 Node.js 的代码也显得更为简洁。对于 libcoap，本章仅介绍如何使用 libcoap 提供的可执行文件，若使用 libcoap 提供的 API，需要自行编写 Makefile 文件，从某种角度来说，libcoap 的使用和学习难度相较于 Java、Python 和 Node.js 更大。

虽然各种 CoAP 框架存在明显区别，但是并不能简单地说在 CoAP 应用领域 Node.js 优于 Python 3，Python 3 优于 Java，Java 优于 C。每种计算机语言、每种 CoAP 框架都有自身的特点和使用场景。Node.js 网络功能强大，通过回调的方式使代码显得异常简洁；Python 2 或者 Python 3 作为树莓派中的默认开发语言，有大量的操作硬件的示例，若 Python aicoap 框架结合树莓派以及外围硬件，可以制作出五花八门的硬件应用；而 Java Californium 虽然稍显复杂，但可以融入到 Android 开发中，那么使用手机也可以通过 CoAP 控制硬件设备，这也留给用户很大的想象空间；C 语言在嵌入式领域中占有不可忽视的地位，多数嵌入式设备仍不具备运行 Java、Python 和 Node.js 的能力，那么 C 语言在嵌入式设备的 CoAP 开发中也会获得广泛的应用。



Chapter 8 第 8 章

CoAP 调试工具

8.1 本章主要内容

本章将重点学习 CoAP 的调试技巧。在 HTTP 应用开发过程中包含很多调试工具和调试技巧，这些调试工具可以帮助用户发送合适的 HTTP 请求，通过这些工具也可以修改 HTTP 请求首部和请求负载，以及验证服务器是否正常工作。例如，使用 Firefox 浏览器中的扩展插件 Httprequester 模拟发送 HTTP 请求；使用 Wireshark 分析 HTTP 请求和 HTTP 响应；使用 cURL 客户端工具模拟发送 HTTP 请求等。在 CoAP 服务的开发过程中也有这种类型的辅助工具，通过这些辅助工具不但可以验证 CoAP 协议的各种细节，还可以帮助用户加速完成 CoAP 服务器和 CoAP 客户端的开发工作。

本章中我们将重点学习两种调试工具——Firefox 中的 Copper 插件和 Wireshark。Copper 插件是一款 CoAP 客户端专用插件；Wireshark 是常用的网络抓包工具，通过该工具不但可以展现 CoAP 的所有细节，还可以了解 CoAP 与其他 TCP/IP 族协议之间的关系。

8.2 Copper 调试工具

Copper 是一款非常容易上手的 CoAP 客户端调试工具，在前面多个章节中已经介绍并使用了该插件，本节我们将更深入地了解 Copper 插件的使用细节。

Copper 插件是一款客户端调试工具，在没有 CoAP 服务器的情况下单独使用 CoAP 客户端是没有任何意义的。若借助 Copper 插件进行 CoAP 实验可使用本书提供的 CoAP 测试服务器，该 CoAP 测试服务器的域名为 coap://wsncoap.org ；也可以使用 Eclipse 项目提供

的 CoAP 测试服务器，该测试服务器的域名为 coap://californium.eclipse.org/。本节并不推荐“生硬”地学习 Copper 插件，而应结合本书第 5 和 6 章，通过 Copper 实验的方式深入理解 CoAP 的细节。在 Firefox 浏览器中安装 Copper 的具体步骤可参考 4.2 节。若正确安装了 Copper 插件，可以在 Firefox 浏览器地址栏中输入“ coap://wsncoap.org”，按回车键之后可观察到如图 8-1 所示的相似界面。

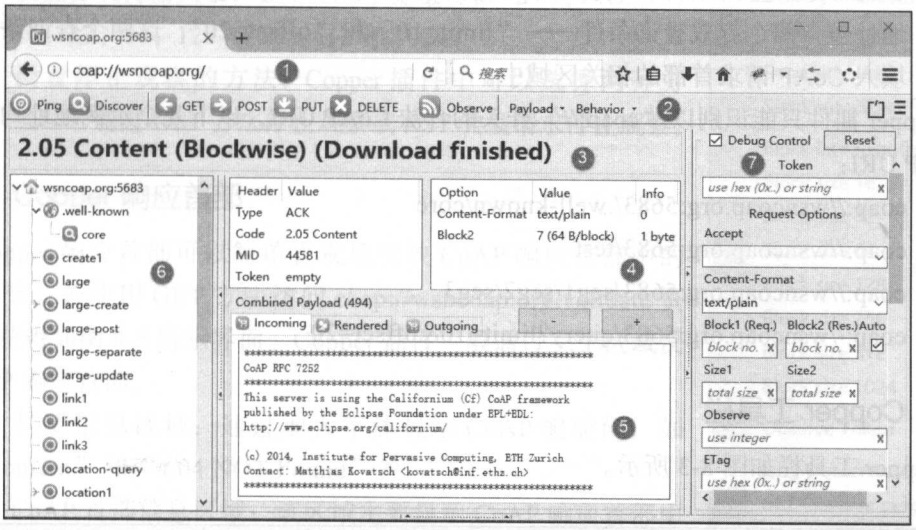


图 8-1　Copper 调试工具

Copper 调试工具大致可以分为以下几个部分：地址栏、工具栏、响应消息简要提示、响应消息首部面板、请求或响应内容面板、路由面板、请求选项设置面板。

8.2.1　Copper 地址栏

Copper 插件与 Firefox 浏览器共用地址栏，Copper 地址栏中的服务器地址必须以“coap://”开头，Copper 插件地址栏的使用方法如图 8-2 所示。

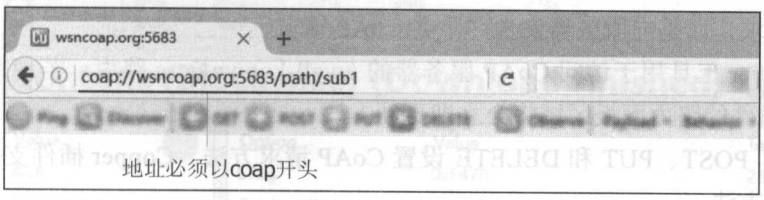


图 8-2　Copper 地址栏

Copper 地址栏可以设置 CoAP 请求中的很多内容，这些内容包括：

- CoAP 服务器域名或 IP 地址：例如此处服务器域名为 wsncoap.org，该服务器的全网 IP

地址为 139.196.187.107，在地址栏中填入 coap://wsncoap.org 与 coap://139.196.187.107 的效果几乎相同。

- CoAP 服务器端口号：例如此处 CoAP 应用端口号为 5683。
- CoAP 资源路由名称：例如 coap://wsncoap.org:5683/seg1/seg2/seg3，其中 seg1/seg2/seg3 将填入 CoAP 请求首部的相关区域中。
- CoAP 资源查询条件：例如 coap://wsncoap.org:5683/query?limit=10&offset=20，该地址包含两个有效查询条件——“limit=10”和“offset=20”，这两个查询条件也会填入 CoAP 请求首部的相关区域中。

Copper 插件只能识别以“coap://”开头的合法 CoAP URI，以下几个示例地址均为合法的 CoAP URI：

- coap://wsncoap.org:5683/.well-known/core
- coap://wsncoap.org:5683/test
- coap://wsncoap.org:5683/seg1/seg2/seg3
- coap://wsncoap.org:5683/query?limit=10&offset=20

8.2.2 Copper 工具栏

Copper 工具栏如图 8-3 所示。

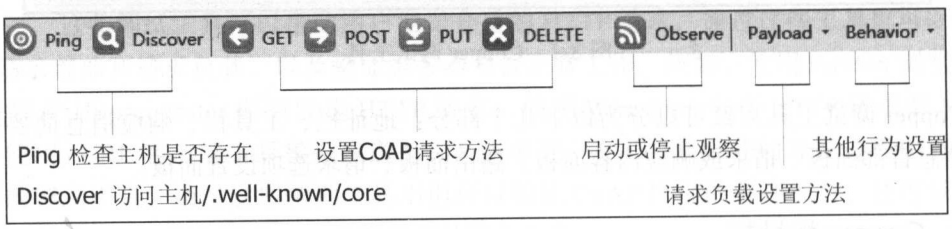


图 8-3 Cooper 工具栏

Copper 工具栏包含以下主要功能：

1）Ping 工具用于测试服务器中的 CoAP 服务是否正常启用。该 Ping 工具并不向主机发送 ICMP 请求，而是向服务器发送一个空 CoAP 请求。

2）Discover 工具用于访问 CoAP 服务器的 /.well-known/core 路由。若使用 Discover 工具，Copper 插件的路由面板区域将会立即更新。

3）GET、POST、PUT 和 DELETE 设置 CoAP 请求方法。Copper 插件支持 CoAP 中规定的 4 种请求方法。

4）Observe 启动或停止观察 CoAP 服务器中的某个资源。

5）Payload 设置 CoAP 请求负载的设置方式，Copper 插件有两种设置请求负载的方法——文本方法和文件方法。默认选择文本方法，选择文件方法时需要指定文件的具体路径。

6）Behavior 设置 Copper 其他常用行为，该工具包含一个下拉菜单，Behavior 工具的具体选项如图 8-4 所示。Behavior 选项内容较多，大致包括以下内容：

- 选择 CoAP 请求类型：默认为 CON 请求，也可以设置为 NON 请求。
- 设置分块传输大小：在 CoAP 中分块传输大小可以设置为 16、32、64、128、256、512 和 1024。
- 设置停止观察的方法：Copper 插件中可以选择两种方法——通过 GET 方法停止观察或 RST 类型请求停止观察。

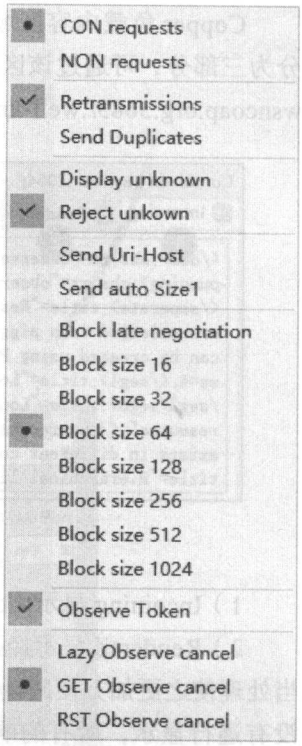


图 8-4　Behavior 选项设置

8.2.3　Copper 响应首部

Copper 响应首部可以简单直观地反映 CoAP 响应首部中的所有细节。若使用 GET 方法访问 coap://wsncoap.org/validate 资源，可获得如图 8-5 所示界面。Copper 插件一般通过三个区域反馈 CoAP 响应首部。

1）简要信息区域：通过加大字体显示 CoAP 响应码，如“2.05 Content”和“4.04 NOT Found”等。

2）CoAP 首部信息区域：该区域主要显示 CoAP 响应首部中的四个固定子内容，如报文类型、响应码、报文编号和报文标签。

3）CoAP 选项信息：该区域主要显示 CoAP 选项部分，如 Etag 选项、Content-Format 媒体类型选项、分块传输选项等内容。与 CoAP 首部信息区域不同，该区域的子项个数不确定，不同的 CoAP 响应所包含的 CoAP 选项数量可能不同。

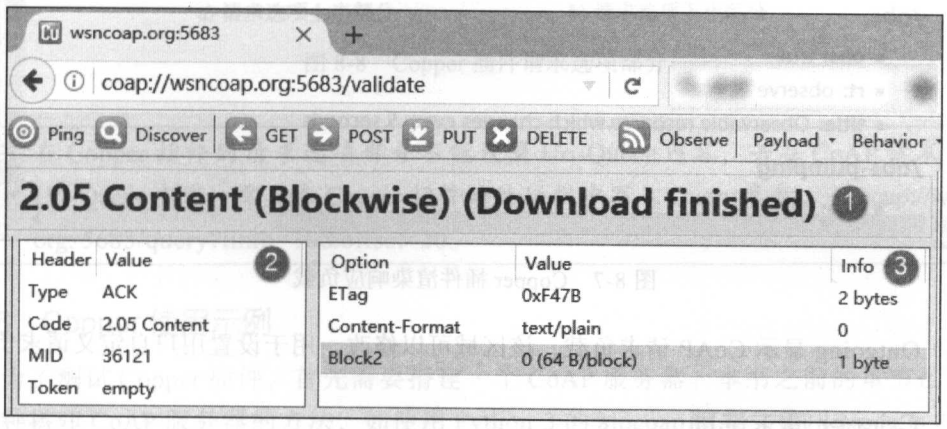


图 8-5　Copper 响应首部

8.2.4 Copper 负载内容

Copper 负载内容区域主要用于显示 CoAP 请求负载和 CoAP 响应负载，该区域也可以分为三部分，可通过该区域上半部分的选项卡标签进行切换。若使用 GET 方法访问 coap://wsncoap.org:5683/.well-known/core，在 Copper 负载部分将会获得如图 8-6 所示界面。

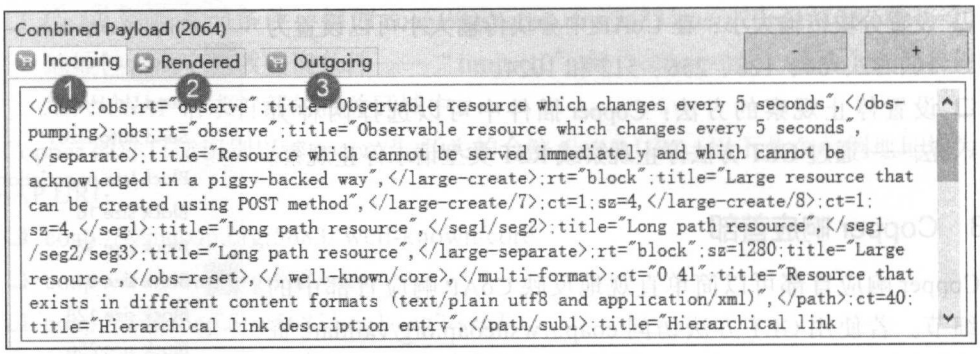


图 8-6 Copper 负载部分

1）Incoming 显示 CoAP 响应文本内容，该区域不可修改。

2）Rendered 显示经过渲染的响应内容，该区域不可修改。该区域把 CoAP 响应内容适当处理使之更加方便阅读与分析。在图 8-6 中 CoAP 响应完全以文本形式展现，该部分甚至没有逐行显示，所有的响应负载被堆砌在一个文本框中。在图 8-7 中响应负载经过渲染处理，响应内容使用更加直观的形式展现，大大简化了阅读与分析难度。Copper 插件能够渲染 link-format 格式、XML 格式和 JSON 格式。

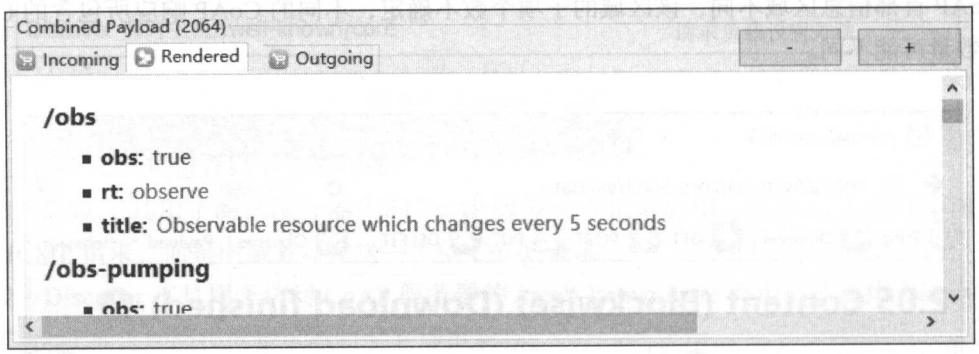


图 8-7 Copper 插件渲染响应负载

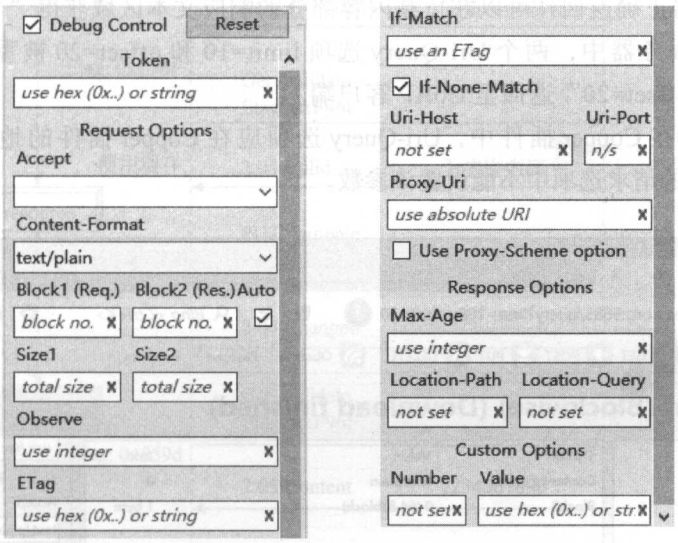
3）Outgoing 显示 CoAP 请求负载，该区域可以修改，用于设置用户自定义请求负载。

8.2.5 Copper 请求选项

Copper 请求选项部分是 Copper 插件中最难掌握的部分，其具体内容如图 8-8 所示。在

CoAP 请求中增加选项内容前需要选中复选框“Debug Control”，否则所有的请求选项均为失效状态。Copper 插件的请求选项较多，它支持的请求选项包括：

- CoAP 请求标签 Token。
- CoAP 请求期望媒体类型设置 Accept。
- CoAP 请求媒体类型设置 Content-Format。
- CoAP 块传输选项 Block1、Block2、Size1 和 Size2。
- CoAP 观察者选项 Observe。
- CoAP 实体标记 Etag。
- CoAP 条件请求选项 If-Match 和 If-None-Match。
- CoAP Uri-Host 和 Uri-Port 等。



a) 请求选项上半部分　　b) 请求选项下半部分

图 8-8　Copper 插件请求选项部分

注意　在 Copper 插件的请求选项部分不能设置 Uri-Query 内容，若在 CoAP 请求中包含 Uri-Query 内容，需要在 Copper 插件的地址栏中写入 Query 参数，如 coap://wsncoap.org: 5683/query?limit=10&offset=20。

8.2.6　Copper 使用示例

为了测试 Copper 插件，首先需要搭建一个 CoAP 服务器。本书之前的章节已经介绍过多种搭建 CoAP 服务器的方法，如使用 Python 3 的 aiocoap 框架、使用 Node.js 的 node-coap 框架和使用 Java 的 Californium 框架。为了更方便地讨论 Copper 插件的使用方法，本

章将使用 coap://wsncoap.org 提供的 CoAP 测试服务器。CoAP 测试服务器提供了多种用于测试目的的 CoAP 资源，通过这些 CoAP 资源用户可以了解 CoAP 的诸多细节，如 CoAP 分离模式、Etag 选项和 CoAP 观察者模式等。

下面通过三个从简到难的示例说明如何使用 Copper 插件，Copper 插件的使用难点在于熟练使用 CoAP 选项。

1. GET With Query

第一个示例将说明如何使用 CoAP Uri-Query 选项，Uri-Query 示例的具体过程如图 8-9 所示。

1）在 Copper 的地址栏中输入 coap://wsncoap.org:5683/query?limit=10&offset=20。

2）单击工具栏中的 GET 按钮，发送 CoAP 请求。

3）若服务器正确返回，可以在负载内容部分的响应文本区域获得“?limit=10&offset=20”。在 CoAP 服务器中，两个 Uri-Query 选项 limit=10 和 offset=20 被重新组合，并还原为“?limit=10&offset=20”返回至 CoAP 客户端。

再次强调，在 Copper 插件中，Uri-Query 选项应在 Copper 插件的地址栏中体现，在 Copper 插件右侧的请求选项中不能设置该参数。

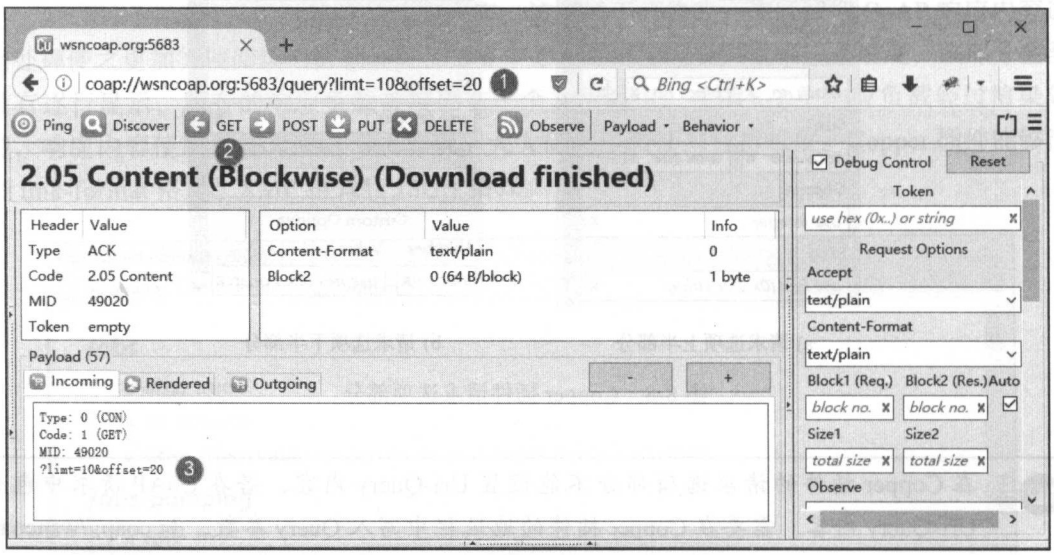


图 8-9 Copper 插件 Uri-Query 选项使用

2. GET /validate With Etag

相比于上一个示例，通过 Etag 选项访问 validate 资源要复杂得多。在该示例中首先通过 GET 方法获取 validate 资源，获取资源的同时记录 Etag；接着使用 GET 方法重新访问 validate 资源，在请求首部中填入 Etag，由于请求首部的 Etag 和服务器中 validate 的 Etag

完全相同，服务器不会重复返回资源，而是通过 2.03 Valid 告知客户端资源并没有改变，可使用上一次的缓存；然后 CoAP 客户端通过 PUT 方法强制改变资源；由于资源被改变，当 CoAP 客户端再次使用缓存的 Etag 时，CoAP 服务器将会返回 validate 的最新内容；最后通过 DELETE 方法删除资源。通过该示例重点学习 Etag 的使用方法和 2.03 Valid 的具体含义。该示例的具体过程如图 8-10 所示。

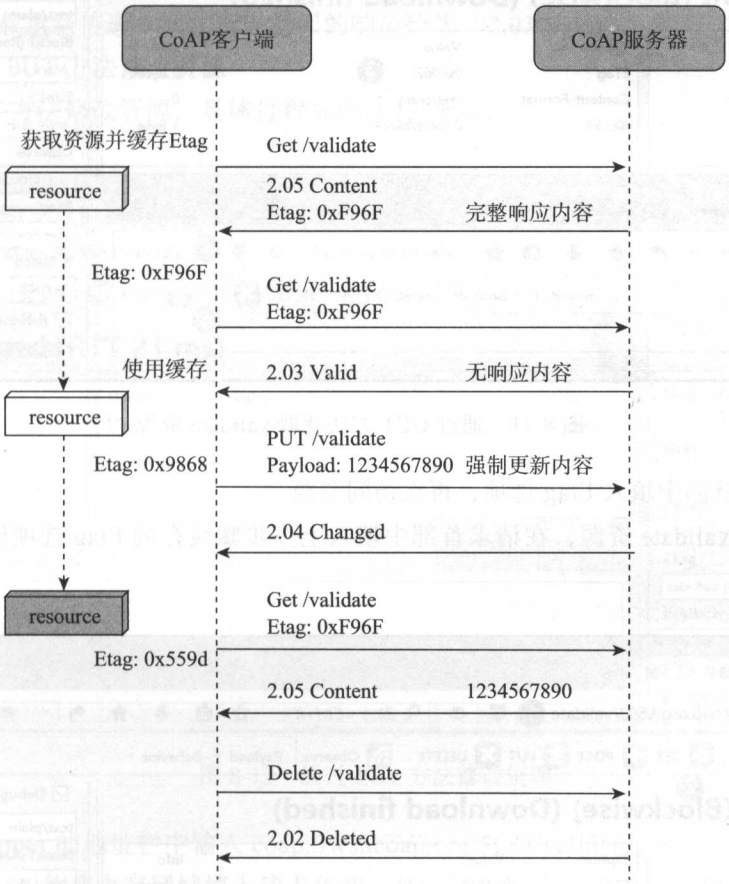


图 8-10　Copper 示例 2 流程说明

（1）通过 GET 方法获取 validate 资源

通过 GET 方法获取 validate 资源，具体过程如图 8-11 所示。

1）在 Copper 的地址栏中输入 coap://wsncoap.org:5683/validate。

2）单击工具栏中的 GET 按钮，发送 CoAP 请求。

3）若服务器正确返回，可在 Copper 插件的 CoAP 选项信息区域获取 CoAP 服务器返回的 Etag，此时返回的 Etag 为“0xF69F”，记录该 Etag 值以便下次使用。

此时 CoAP 服务器返回的响应码为“2.05 Content”，响应负载中包含具体内容。

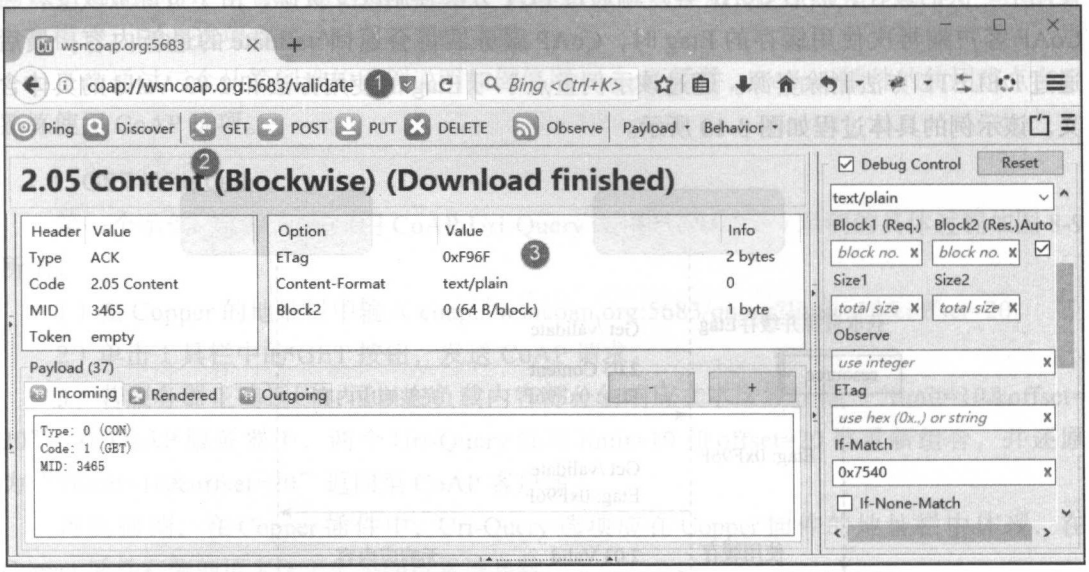


图 8-11　通过 GET 方法获取 validate 资源

（2）请求首部中填入 Etag 选项，再次访问资源

再次访问 validate 资源，在请求首部中填入上一步骤缓存的 Etag 选项值，具体过程如图 8-12 所示。

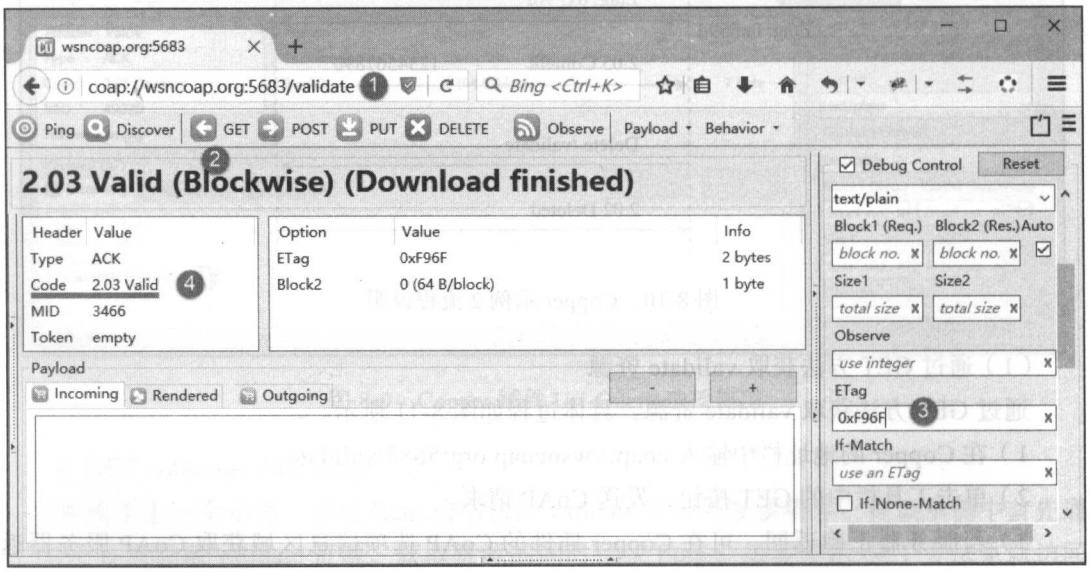


图 8-12　请求首部中填入 Etag 选项，再次访问资源

1）在 Copper 的地址栏中输入 coap://wsncoap.org:5683/validate。

2）在 Copper 选项区域的 Etag 参数部分填入 0xF96F。

3）单击工具栏中的 GET 按钮，发送 CoAP 请求。

4）若服务器正确返回，在 CoAP 首部信息区域可观察到 CoAP 响应码为“2.03 Valid”，而且 CoAP 响应中没有任何负载内容。

此时服务器认为 CoAP 客户端中缓存的负载与服务器中的负载完全相同，没有必要把完全相同的负载重新传递一次，所以返回的响应码为“2.03 Valid”。

（3）通过 PUT 方法修改资源

通过 PUT 方法修改资源，具体过程如图 8-13 所示。

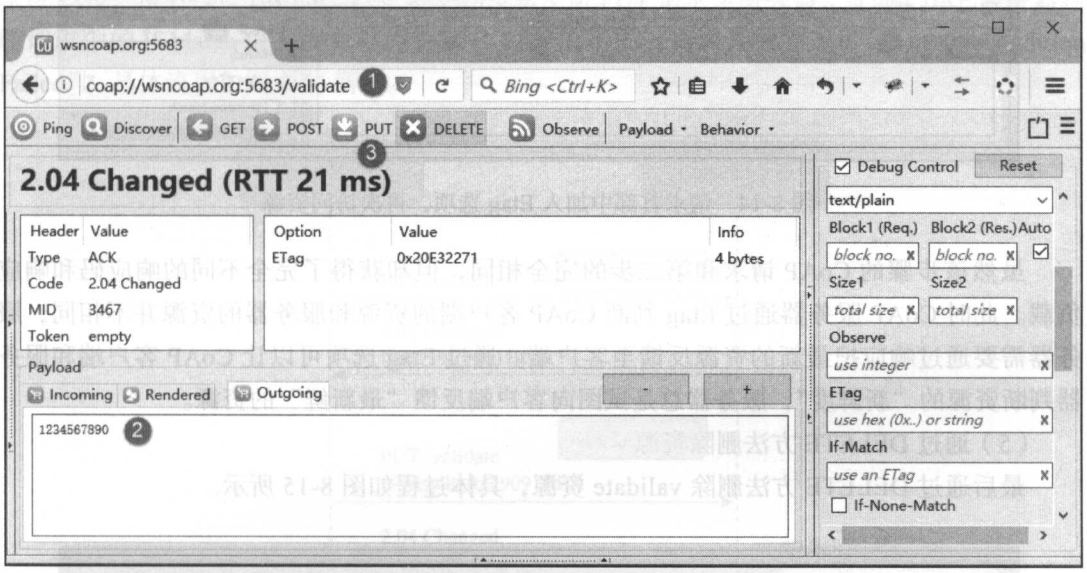


图 8-13　通过 PUT 方法修改资源

1）在 Copper 的地址栏中输入 coap://wsncoap.org:5683/validate。

2）在 CoAP 负载内容区域填入请求负载，如字符串形式的“1234567890”。

3）单击工具栏中的 PUT 按钮，发送 CoAP 更新请求。

（4）请求首部中加入 Etag 选项，再次访问资源

下面试图通过 GET 方法再次获取 validate 资源，此处填入与第二步完全相同的 Etag 参数，如图 8-14 所示。

1）在 Copper 的地址栏中输入 coap://wsncoap.org:5683/validate。

2）在 Copper 选项区域的 Etag 参数部分填入 0xF96F。

3）单击工具栏中的 GET 按钮，发送 CoAP 请求。

4）若服务器正确返回，在 CoAP 首部信息区域可观察到 CoAP 响应码为“2.05 Content”，

而且 CoAP 响应中包含具体内容“1234567890”。

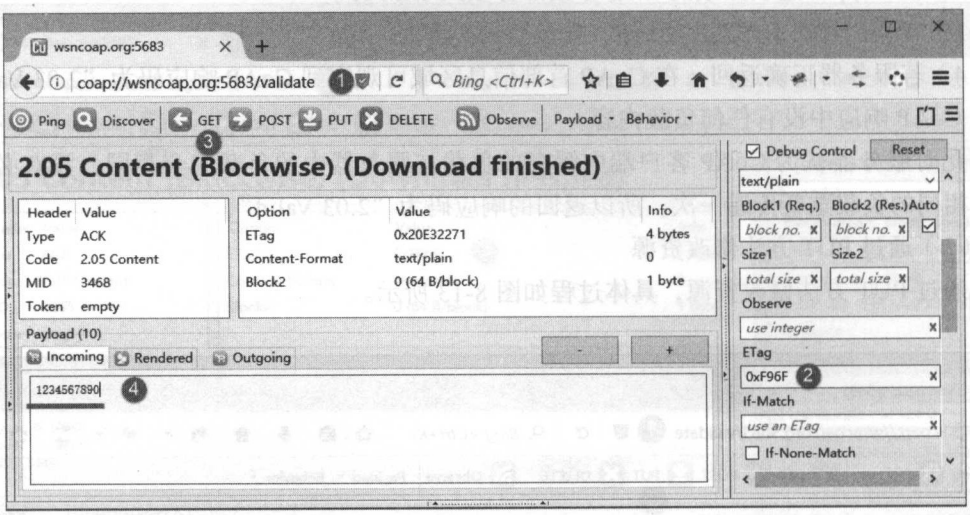


图 8-14　请求首部中加入 Etag 选项，再次访问资源

虽然该步骤的 CoAP 请求和第二步的完全相同，但却获得了完全不同的响应码和响应负载。此时 CoAP 服务器通过 Etag 判断 CoAP 客户端的资源和服务器的资源并不相同，服务器需要通过响应把最新的资源反馈至客户端。通过 Etag 选项可以让 CoAP 客户端和服务器判断资源的“新鲜度”，服务器总是试图向客户端反馈“最新鲜”的资源。

（5）通过 DELETE 方法删除资源

最后通过 DELETE 方法删除 validate 资源，具体过程如图 8-15 所示。

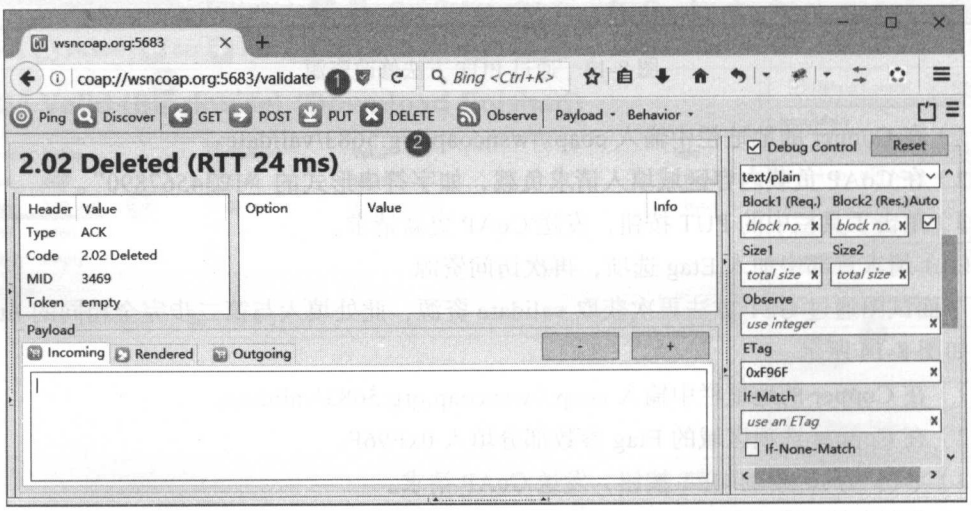


图 8-15　通过 DELETE 方法删除资源

1）在 Copper 的地址栏中输入 coap://wsncoap.org:5683/validate。

2）单击工具栏中的 DELETE 按钮，发送 CoAP 请求。

3）若服务器正确返回，将在 Copper 插件的响应首部区域获取“2.02 Deleted”响应码。

3. PUT With If-Match

在本示例中首先在 Copper 插件的请求负载区域填入任意内容，通过 PUT 方法试图更新 validate 资源，更新资源之后将获得一个 Etag，在该示例中我们将记录该 Etag；接着在 Copper 插件的请求负载区域填入与上一步不完全相同的内容，在 Copper 请求选项区域 If-Match 选项中填入上一步记录的 Etag 值，通过 PUT 方法再次更新 PUT 请求，此时服务器的响应码为“2.04 Changed”，并且把更新之后的 Etag 值反馈至客户端，但是此时我们并不会记录该 Etag；最后保持 If-Match 参数不变，重新发送 PUT 请求至服务器，此时 CoAP 服务器判断请求中的 If-Match 值与服务器的资源的 Etag 值不同，请求无效并返回“4.12 Precondition Failed”。具体流程如图 8-16 所示。

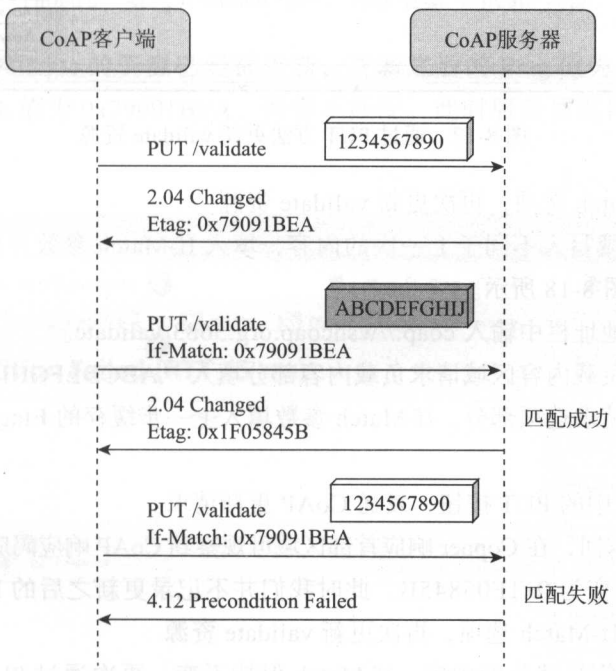


图 8-16　Copper 插件示例 3 流程说明

（1）通过 PUT 方法更新 validate 资源

通过 PUT 方法更新 validate 资源，具体过程如图 8-17 所示。

1）在 Copper 地址栏中输入 coap://wsncoap.org:5683/validate。

2）在 Copper 负载内容区域请求负载内容部分填入“1234567890”。

3）单击工具栏中的 PUT 按钮，发送 CoAP 更新请求。

若服务器正确返回，在 Copper 响应首部区域可观察到 CoAP 响应码为“2.04 Changed”，而且响应中 Etag 的值为 0x79091BEA，记录该 Etag 的值并填入下一步的 If-Match 选项中。

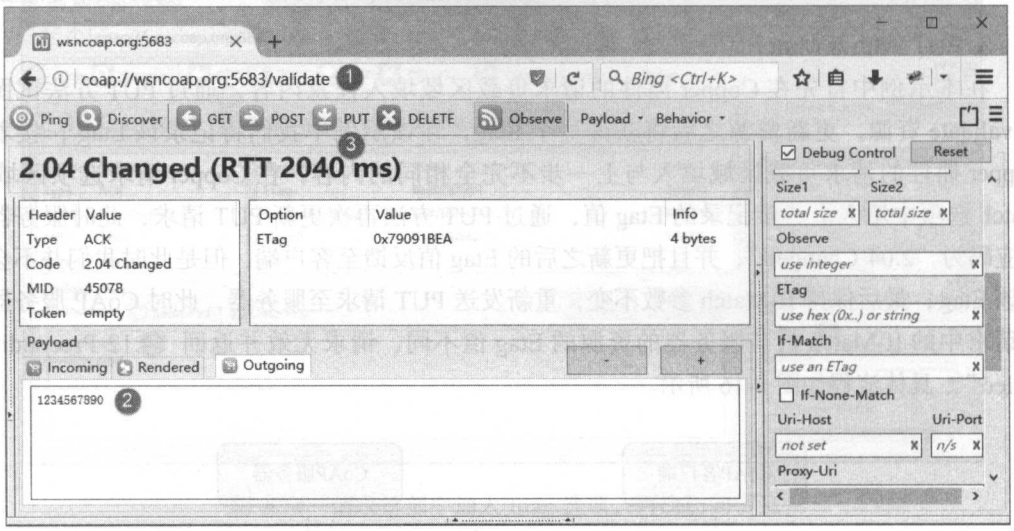


图 8-17　通过 PUT 方法更新 validate 资源

（2）填入 If-Match 选项，再次更新 validate 资源

在请求负载区域写入不同于上一次的内容，填入 If-Match 参数并再次发送 CoAP 更新请求，具体过程如图 8-18 所示。

1）在 Copper 地址栏中输入 coap://wsncoap.org:5683/validate。

2）在 Copper 负载内容区域请求负载内容部分填入“ABCDEFGHIJ”。

3）在 Copper 请求选项部分，If-Match 参数填入上一步缓存的 Etag 值，此时 If-Match=0x79091BEA。

4）单击工具栏中的 PUT 按钮，发送 CoAP 更新请求。

若服务器正确返回，在 Copper 响应首部区域可观察到 CoAP 响应码应为“2.04 Changed”，但响应中 Etag 的值变为 0x1F05845B，此时我们并不记录更新之后的 Etag。

（3）依然填入 If-Match 选项，再次更新 validate 资源

最后更新请求负载部分的内容，If-Match 保持不变，再次通过 PUT 请求更新 validate 资源。由于 If-Match 参数没有得到及时更新，所以 CoAP 服务器将会拒绝本次 PUT 请求。具体过程如图 8-19 所示。

1）在 Copper 地址栏中输入 coap://wsncoap.org:5683/validate。

2）在 Copper 负载内容区域请求负载内容部分填入“1234567890”或其他任意值。

3）在 Copper 请求选项部分，If-Match 参数填入第一步保存的 Etag 值，此时 If-Match=0x79091BEA。

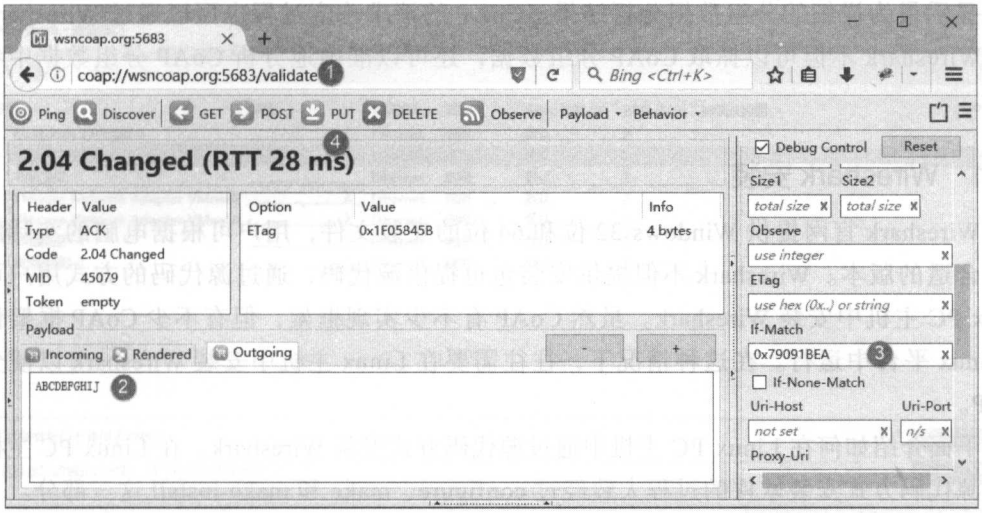


图 8-18　填入 If-Match 选项，再次更新 Validate 资源

在第二步中，validate 的资源已经被更新，更新之后的 Etag 值为 0x1F05845B，而此时请求中的 If-Match 值为 0x79091BEA，两者不匹配。此时服务器返回的响应码为“4.12 Precondition Failed”。

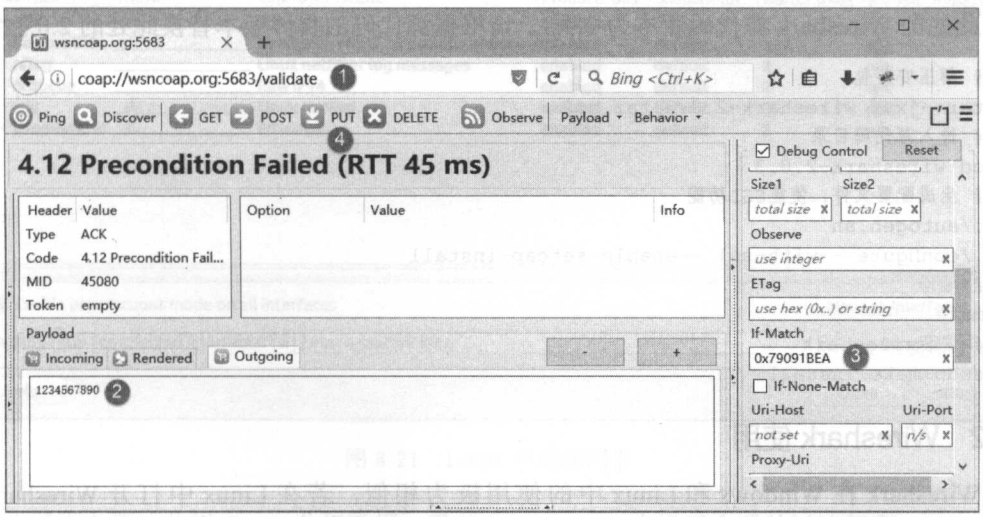


图 8-19　依然填入 If-Match 选项，再次更新 Validate 资源

8.3　Wireshark

Wireshark 是一款网络抓包分析软件。Wireshark 获取经过网卡的网络分组数据，并尽

可能显示最为详细的分组数据分析结果。CoAP 的请求响应过程也可以通过 Wireshark 抓取，Wireshark 不但可以抓取 CoAP 分组数据，还可以准确地分析 CoAP 分组数据中各种细节。

8.3.1 Wireshark 安装

Wireshark 官网提供 Windows 32 位和 64 位的安装文件，用户可根据电脑的实际配置安装合适的版本。Wireshark 不但提供安装包也提供源代码，通过源代码的方式用户可在 Linux PC 主机中安装 Wireshark。虽然 CoAP 有不少实现框架，但有不少 CoAP 框架仅能在 Linux 平台中运行。在这种情况下，往往需要在 Linux 主机中安装 Wireshark 以便分析 CoAP。

下面介绍如何在 Linux PC 主机中通过源代码方式安装 Wireshark。在 Linux PC 主机中通过源代码方式安装软件的过程大致经过 configure、make 和 make install 这三部分。一般情况下 make install 需要把编译生成的库文件和可执行文件复制到根目录指定位置，所以往往需要超级权限。安装 Wireshark 之前需要在 Linux 主机中安装必要的依赖包，由于每一个 Linux 发行版都不完全相同，用户需要根据实际情况安装必要的依赖包。

```
sudo apt-get build-dep wireshark
sudo apt-get install qt5-default
sudo apt-get install libssl-dev
```

此处的 Wireshark 源代码版本为 2.0.3，请根据源代码的具体版本替换此处的 2.0.3。

```
# 解压安装包
tar -jxvf wireshark-2.0.3.tar.bz2
# 进入源代码目录
cd wireshark-2.0.3
# 生成配置文件，使能SSL功能
./autogen.sh
./configure --with-ssl --enable-setcap-install
# 编译
make
# 安装wireshark
sudo make install
```

8.3.2 Wireshark 使用

Wireshark 在 Windows 和 Linux 中的使用极为相似，若在 Linux 中打开 Wireshark 可先新建一个控制台并在控制台中输入 wireshark 保持控制台处于运行状态即可。开始抓取 CoAP 数据之前需要设置正确的网卡，请根据实际情况选择有线网卡或无线网卡。打开 Wireshark 之后，选择“菜单栏→捕获→选项”。Windows 中网卡的名称和 Linux 中的略有差别，Windows 中有线网卡的名称为“本地连接”（见图 8-20），而 Linux 中有线网卡的名称为“eth0”（见图 8-21）。

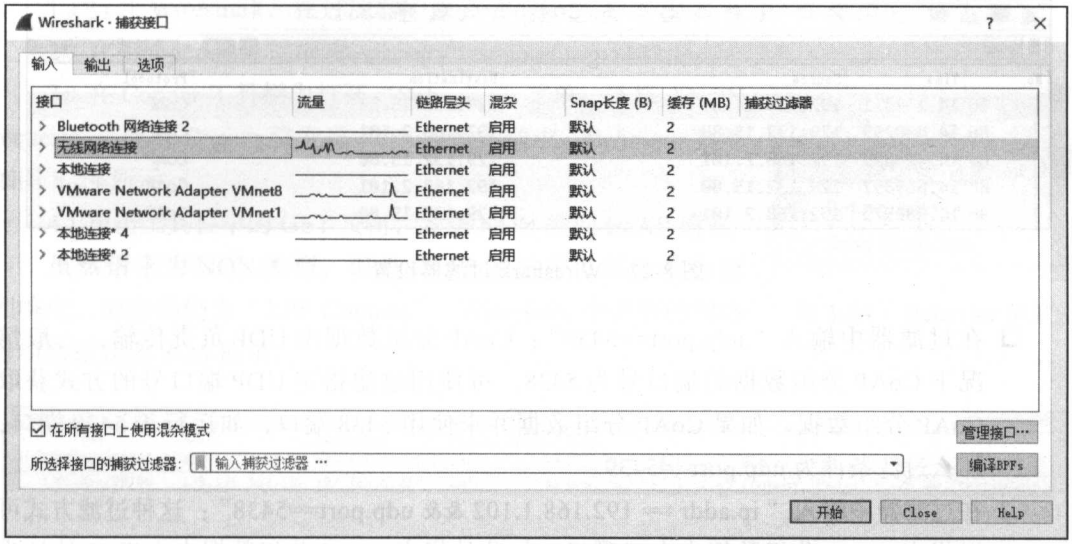


图 8-20　Windows 中选择网卡

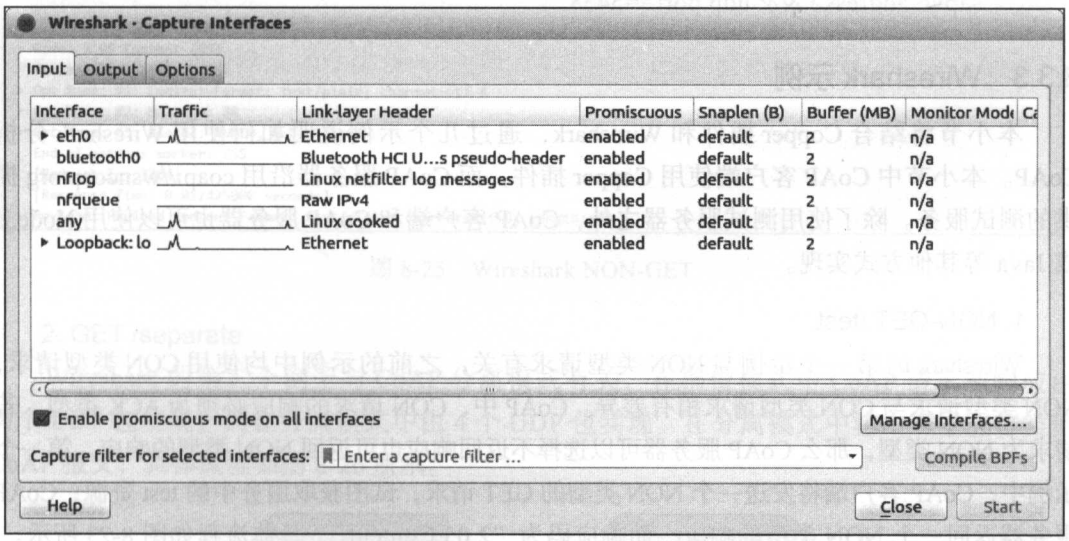


图 8-21　Linux 中选择网卡

为了更方便地分析 CoAP 分组数据，需要在 Wireshark 中设置合适的过滤规则。Wireshark 的过滤器设置位于工具栏下方。如图 8-22 所示。

可通过多种方式过滤 CoAP 分组报文，这些方法包括：

- 在过滤器中输入“coap”：Wireshark 将过滤所有 CoAP 分组数据，这些分组数据包括 CoAP 请求、CoAP 响应等。

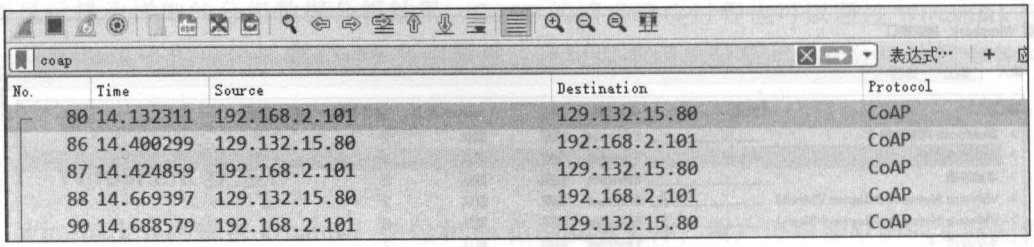


图 8-22 Wireshark 过滤器设置

- 在过滤器中输入“udp.port==5438”：CoAP 分组数据由 UDP 负责传输，一般情况下 CoAP 分组数据的端口号为 5438，可使用过滤指定 UDP 端口号的方式获取 CoAP 分组数据。如果 CoAP 分组数据并未使用 5438 端口，如运行于 5439 端口，那么过滤条件为 udp.port==5439。
- 在过滤器中输入“ip.addr == 192.168.1.102 && udp.port==5438”：这种过滤方式可以指定 CoAP 分组数据来自于哪里，如此处指定 CoAP 分组数据来自于 IP 地址为 192.168.1.102 的主机。若需要过滤 IPv6 地址，可在过滤器中输入“ipv6.address==<ipv6 address> && udp.port==5438”。

8.3.3 Wireshark 示例

本小节将结合 Copper 插件和 Wireshark，通过几个示例说明如何使用 Wireshark 分析 CoAP。本小节中 CoAP 客户端使用 Copper 插件，而 CoAP 服务器沿用 coap://wsncoap.org 提供的测试服务。除了使用测试服务器之外，CoAP 客户端和 CoAP 服务器也可以使用 Node.js 或 Java 等其他方式实现。

1. NON-GET /test

Wireshark 的第一个示例与 NON 类型请求有关，之前的示例中均使用 CON 类型请求，NON 类型请求与 CON 类型请求稍有差异。CoAP 中，CON 请求的响应必须为 ACK 类型，若请求为 NON 类型，那么 CoAP 服务器可以选择不返回响应也可返回 NON 类型的响应。第一个示例中，CoAP 客户端将发送一个 NON 类型的 GET 请求，试图获取服务中的 test 资源；CoAP 服务器返回一个 NON 类型的响应，而响应码为“2.05 Content”，具体流程如图 8-23 所示。

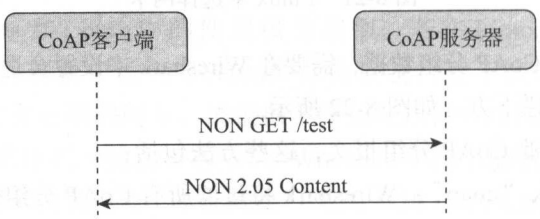


图 8-23 NON GET 请求

1）打开 Wireshark，在过滤器中输入“coap”。

2）在 Copper 地址栏中输入 coap://wsncoap.org:5683/test。

3）在 Copper 工具栏中打开“Behavior”菜单，选择“NON requests”，此时 Copper 将发送 NON 类型的请求。Behavior 菜单如图 8-24 所示。

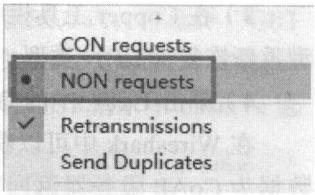


图 8-24　Behavior 菜单

4）单击工具栏中的 GET 按钮，发送 NON 类型 CoAP 请求。

虽然请求为 NON 类型，但是 CoAP 测试服务器依然会返回响应，响应码仍为“2.05 Content”。Wireshark 中获取的网络分组数据如图 8-25 所示。

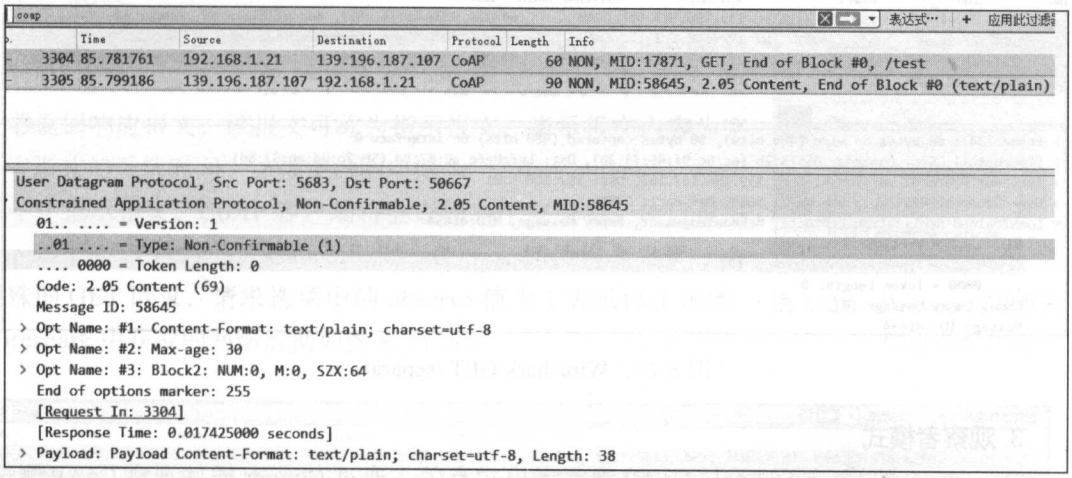


图 8-25　Wireshark NON-GET

2. GET /separate

Wireshark 的第二个例子与 CoAP 分离模式有关，在携带模式中 CoAP 请求和响应由两个 UDP 包实现，而在分离模式中由 4 个 UDP 包实现。在分离模式中将出现空应答类型 CoAP 报文，具体流程如图 8-26 所示。

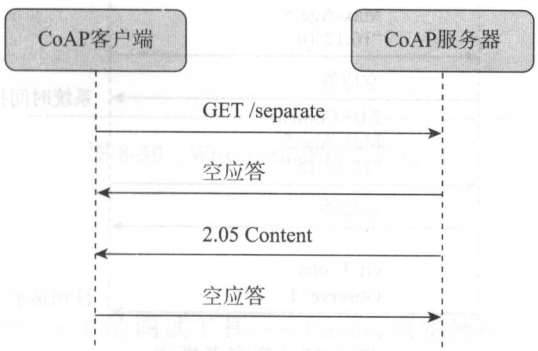


图 8-26　GET /separate

1）重新打开 Wireshark，在过滤器中输入“coap”。

2）在 Copper 地址栏中输入 coap://wsncoap.org:5683/separate。

3）在 Copper 工具栏中打开“Behavior”菜单，重新选择“CON requests”把请求类型重新恢复为 CON 类型。

4）单击工具栏中的 GET 按钮，发送 CoAP 请求。

在 Wireshark 中可以获取 4 组网络数据，其中第二组网络分组数据较为特殊，该组网络数据为 CoAP 服务器返回至 CoAP 客户端的“空应答”，在 CoAP 中空应答的 Code 区域的值为 0x00。Wireshark 中获取的网络分组数据如图 8-27 所示。

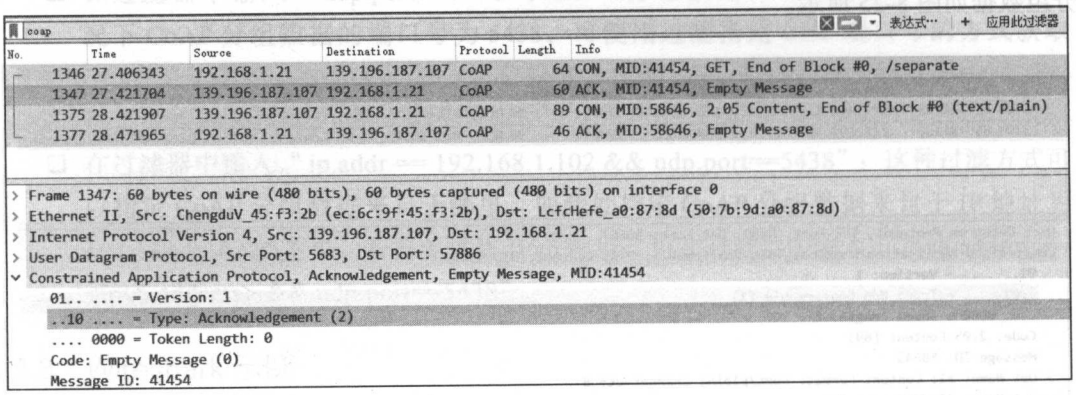


图 8-27　Wireshark GET /separate

3. 观察者模式

Wireshark 的第三个例子与 CoAP 观察者模式有关。通过 Copper 插件观察 CoAP 测试服务器中的 obs 资源，观察一段时间之后再停止观察。具体流程如图 8-28 所示。

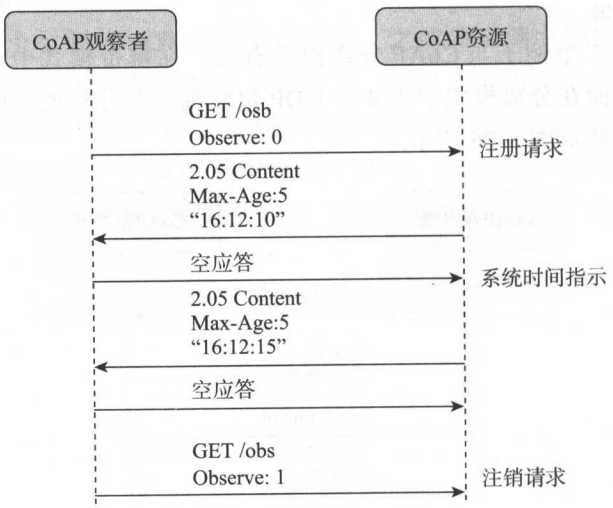


图 8-28　观察者模式

1）重新打开 Wireshark，在过滤器中输入“coap”。

2）在 Copper 地址栏中输入 coap://wsncoap.org:5683/obs。

3）在 Copper 工具栏中打开“Behavior”菜单，选择停止观察模式方式，可选择“GET Observe Cancel”，Behavior 菜单如图 8-29 所示。若选择“GET Observe Cancel”，Copper 插件通过发送一个特殊的 GET 请求以停止观察者模式，在该特殊的 GET 请求中 observe 选项值为 1。

4）单击工具栏中的 Observe 按钮，启动观察。一旦启动观察，Observe 按钮将变为 Cancel 按钮。

5）经过一段时间之后点击工具栏中的 Cancel 按钮，以停止观察者模式。

通过 Wireshark 可以获得多组网络数据，包括用于启动观察的 GET 请求，该 GET 请求中 observe 选项的值为 0；CoAP 服务器返回响应报文，该报文可称为指示报文，指示报文内容为字符串形式的系统时间，该报文的响应码为“2.05 Content”；对于每一个指示报文，CoAP 客户端将会返回一个空应答；最后点击 Copper 插件工具栏的“Cancel”时，Copper 插件将会发送一个特殊的 GET 请求，请求选项中的 observe 值为 1 表示停止观察。Wireshark 中获取的网络数据如图 8-30 所示。

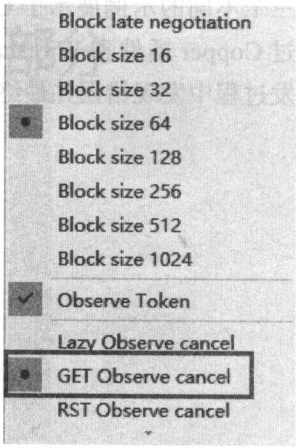


图 8-29　Behavior 菜单修改停止观察方式

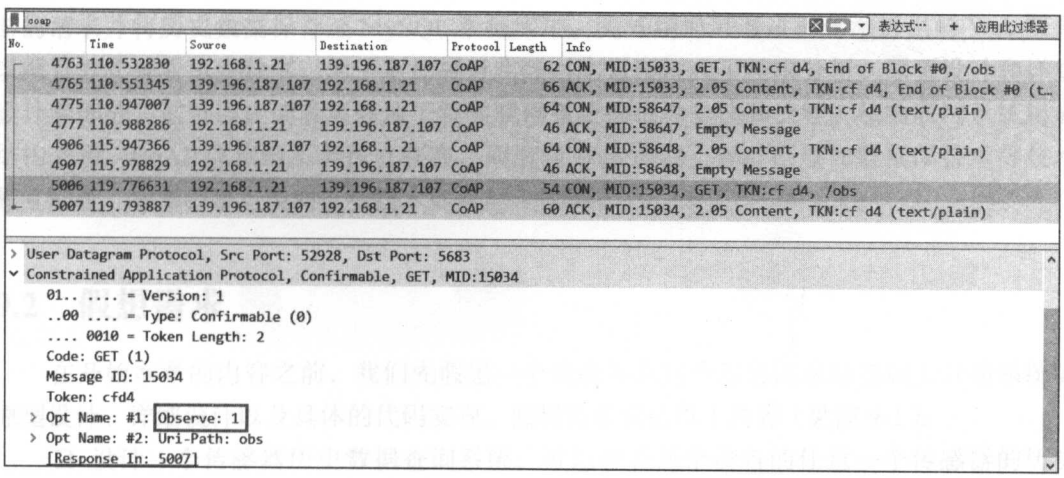


图 8-30　Wireshark 观察者模式

8.4　本章小结

本章主要介绍了两种 CoAP 的调试工具——Firefox 浏览器中的 Copper 插件和网络嗅

探器 Wireshark，本质上来说，CoAP 的调试技巧与 HTTP 的调试技巧非常相似。Copper 插件并不是 CoAP 全能工具，它只是一款 CoAP 客户端工具。在 Copper 插件介绍部分详细说明了如何设置 CoAP URI，如何填充 CoAP 请求负载，如何设置 CoAP 条件请求参数，如何观察 CoAP 响应码等。在 Wireshark 部分，介绍了如何在 Windows 平台和 Linux 平台安装 Wireshark，并详细介绍了如何过滤 CoAP 报文的三种不同方法；在 Wireshark 部分也通过三个不同的示例展示了 CoAP NON 类型请求、分离模式和观察者模式的具体工作流程。通过 Copper 插件和 Wireshark 工具的配合使用，可以帮助用户了解 CoAP 细节，在实际的开发过程中发现错误并最终解决问题。

第 9 章 Chapter 9

微型物联网系统——服务器部分

9.1 本章主要内容

本章将使用树莓派 3 代实现一个更为接近真实场景的应用。在这个应用中，树莓派作为服务器，此处树莓派不但提供 CoAP 服务，还提供 HTTP Web 服务。树莓派将接收传感器的请求并将历史数据保存至 MySQL 数据库中，另外用户可通过树莓派提供的 Web 服务以网页方式查看历史数据。为了更加接近实际，本章还加入了需求分析、原型设计和详细设计等部分，这可以让读者更好地了解物联网系统的设计方法和流程。从物联网系统组成结构来看，CoAP 应用绝不会孤立存在，而应与 Web 前端、Web 后端和数据库技术等技术相互配合。本章将重点说明这些技术如何配合使用。

9.2 假想需求

在开始本章的内容之前，我们先假想一个需求并在这个假想需求的基础上开始系统的原型设计、详细设计以及具体的代码实现。假想需求包括以下内容（见图 9-1）：

- 设计一个传感器历史数据查询系统，可以查看每个设备的任意一个传感器的历史数据。
- 单个设备具有三个传感器：温度传感器、湿度传感器和光照传感器。
- 单个设备将定时推送传感器检测结果。
- 服务器可同时接收多个设备请求，解析设备请求内容之后保存至数据库中。
- 具有设备管理与查询界面，可通过 Web 页面查看设备信息、历史信息。

- 设备管理与查询界面美观大方，便于用户使用。

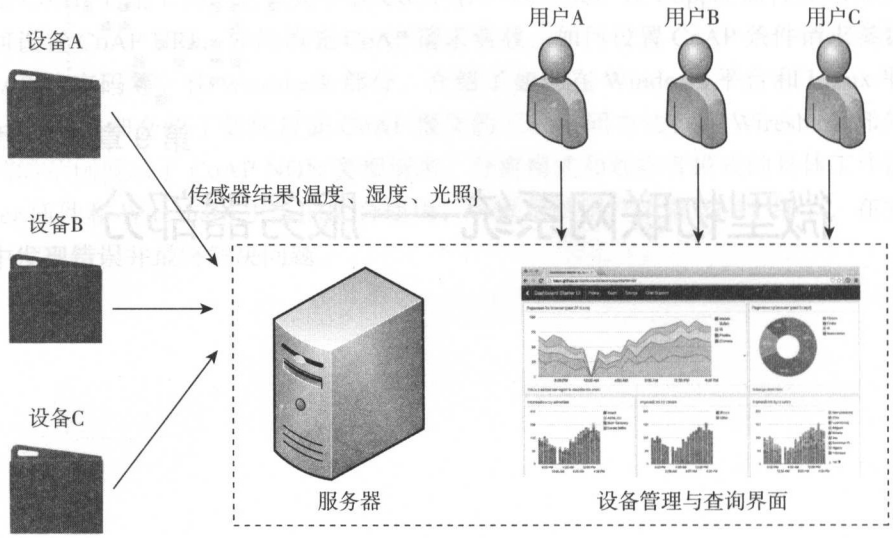


图 9-1 假想需求说明

9.3 原型设计

在了解系统需求之后我们便可着手开始后续的原型设计。原型设计可以帮助工程师了解该做些什么、系统有哪些功能等基本问题。大多数物联网系统都包含很多部分，这些部分可能包括 CoAP 部分、Web 前端、Web 后端和数据库，甚至还可能包括数据分析和机器学习等内容。获取物联网系统的需求之后，应该着手进行必要的设计工作，此时开始写代码并实现具体功能绝对不是一个好“主意”。

9.3.1 系统结构说明

若从具体的需求入手，那么该物联网系统大致可以分为 3 个部分——设备交互部分、Web 系统部分、数据库部分。系统的组成结构如图 9-2 所示。

（1）设备交互部分

设备交互部分负责接收传感器请求内容，CoAP 可以非常好地胜任该部分工作，CoAP 使设备可以方便地接入传统互联网，CoAP 具有很多 HTTP 的特性，这也使得系统开发变得更容易。

（2）Web 系统部分

物联网系统具有一个设备管理和查询界面，该部分是一个典型的 Web 系统，而 Web 系统设计一般可分为前端设计和后端设计两部分。Web 前端负责展现页面，提供必要人机交

互操作；Web 后端负责处理所有的请求路由。

（3）数据库部分

对于绝大多数物联网或者 Web 系统来说，数据保存与查询是不可或缺的部分，该系统需要及时保存设备上传的传感器数据。

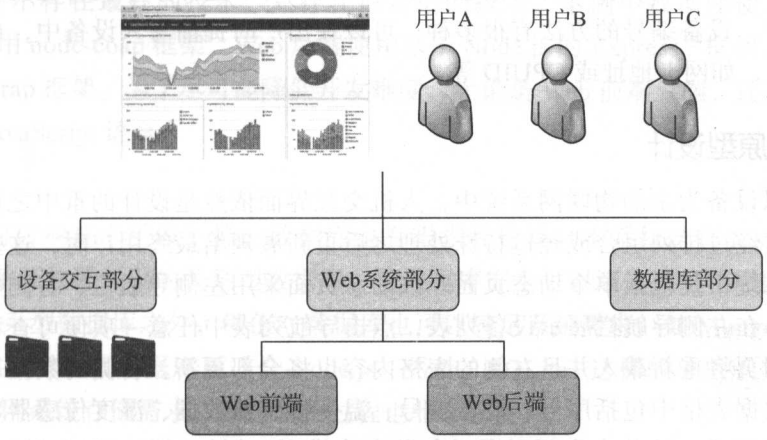


图 9-2　系统结构组成说明

9.3.2　系统流程设计

对系统组成有一个明确的认识之后，便可着手规范系统运行过程中的相关流程。与多数传统的 Web 系统不同，物联网系统中总是有设备参与。在一些移动 Web 系统中，也存在移动设备的概念，但是此处的移动设备多指智能手机，这些设备的运算和存储能力远远超过那些物联网设备。在系统流程设计中应该适当“迁就”物联网设备，简化设备的交互过程。流程设计主要表现设备、用户和系统之间相互操作的关系。微型物联网系统工作流程如图 9-3 所示。

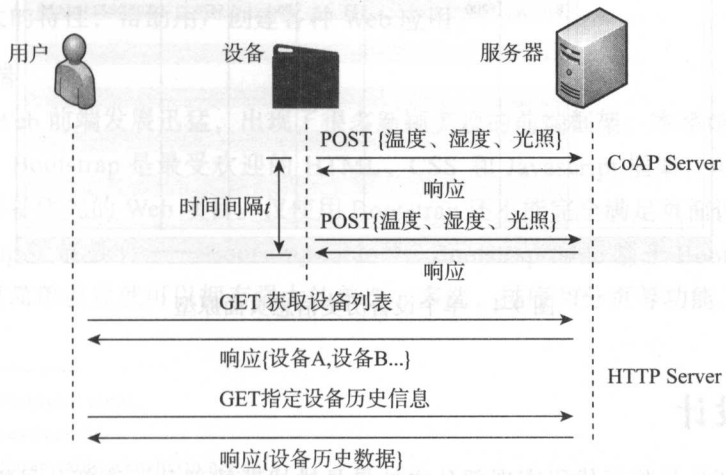


图 9-3　系统流程设计

在该物联网系统中，设备根据间隔时间 t 周期性地推送传感器结果，传感器结果包含一组温度、湿度和光照检测数据。设备的工作流程相对比较简单，而用户的交互流程要稍微复杂一些，用户可以获得物联网系统中包含了哪些设备，物联网系统可以返回一个设备列表。另外，用户还可以选择一个具体的设备查看其历史信息。该流程中用户需要先获取设备列表，再获取单个设备的历史信息。最后，用户可以选择单个设备也就意味着设备具有唯一的编号。设备编号的方法有很多种，可以在出厂时提前写入设备中，也可以使用设备本身的信息，如网卡地址或 CPUID 等。

9.3.3 网页原型设计

即使在以设备为主的物联网系统中，人机交互界面依然是设计的重中之重，因为只有当传感器数据经过排列组合或经过特殊处理之后重新展现给最终用户时，这些数据才体现出价值。本系统中仅包括单个动态页面。该动态页面采用左侧导航栏、右侧数据表格这样的布局方式。在左侧导航栏展示设备列表，点击导航列表中任意一项便可查看具体设备的历史信息，网页将重新载入并且右侧的表格内容也将全部更新。右侧的数据表格展示相关历史数据，数据表格中包括序号、设备编号、温度传感器数据、湿度传感器数据、光照数据和上报时间记录，这些内容可以通过各种方式排列，例如可以按照上报时间逆序排列。如果数据内容太多，数据表格应加入分页功能。动态网页的原型如图 9-4 所示。

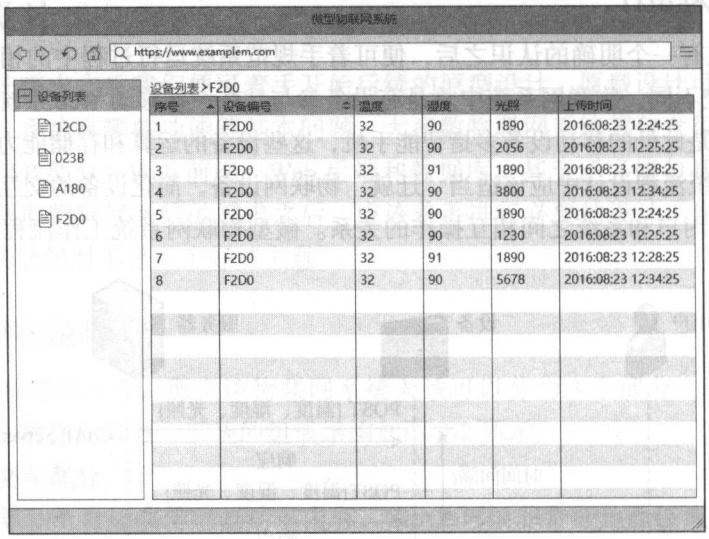


图 9-4 单个设备历史信息页面原型

9.4 详细设计

详细设计部分是除了代码实现部分之外最具挑战性的部分，该部分是物联网系统实现

过程中最重要的部分。

9.4.1　技术选型说明

技术选型是微型物联网系统详细设计的第一步，技术选型往往是一个协调和妥协的工作，世界上并不存在最好的技术，只存在最合适的技术。本例中数据库使用 MySQL[㊀]，CoAP 服务使用 node-coap 框架，Web 后端使用基于 Node.js 的 Express[㊁]框架，Web 前端主要使用 Bootstrap 框架。为了尽可能降低开发难度，无论是 Web 前端后端，还是 CoAP 服务器，均使用 JavaScript 语言。

1. 数据库

与 Web 前后端技术手段一样，数据库也有很多种选择。本系统选择 MySQL 数据库，MySQL 是一款开放源代码的关系型数据库，该数据库完全可以满足微型物联网系统的存储需求。除了关系型数据库之外，键值对数据库也可以满足系统需求，如 MongoDB。从开发角度来看，键值对形式数据库更适合 Node.js。另外，SQLite3 这样的单机版数据库也可以满足微型物联网系统存储需求，而且使用过程也非常方便。

2. CoAP 服务

通过前面内容可以发现目前有诸多 CoAP 实现框架，如 Python 3 的 aiocoap、Node.js 的 node-coap、Java 的 Californium。此处选择 node-coap 实现 CoAP 服务器，Python 相对于 Node.js 而言具有更好的设备操作能力，在树莓派中可使用 Python 控制各种设备或传感器，但是本章系统侧重于 CoAP 服务功能，所以 Node.js 略胜一筹。

3. Web 后端

Web 后端同样采用 Node.js 实现。关于 Web 开发 Node.js 也有不少后端框架，本系统选择 Express 框架。Express 是一个基于 Node.js 平台的极简、灵活的 Web 应用开发框架，它提供一系列强大的特性，帮助用户创建各种 Web 应用。

4. Web 前端

最近几年 Web 前端发展迅猛，出现了很多新颖美观的前端框架。本系统选择 Bootstrap 作为前端框架。Bootstrap 是最受欢迎的 HTML、CSS 和 Javascript 框架，可用于开发响应式布局、移动设备优先的 Web 项目。仅使用 Bootstrap 还不能完全满足页面设计需求，还需要借助 Bootstrap 扩展插件——Bootstrap-table[㊂]。Bootstrap-table 基于 Bootstrap 的 jQuery 表格插件，通过简单设置就可以拥有强大的单选、多选、排序和分页等功能。

㊀ https://www.mysql.com/。

㊁ http://expressjs.com/。

㊂ http://bootstrap-table.wenzhixin.net.cn/。

9.4.2 数据库设计

微型物联网系统并没有用户权限管理，需求中也没有指定相应的角色和职责，所以系统的数据库设计显得简单得多。传感器的历史数据仅需要一张表便可满足存储需求。数据库的名称为 monitor，传感器历史数据表的名称为 sensor_history。传感器历史数据表的具体内容见表 9-1。

表 9-1 传感器历史数据表设计

项　目	变 量 名	类　型	是否为空	说　明
序号	id	int	Not Null	主键
设备编号	device_id	char	Not Null	设备编号 别名设备地址
温度	temp	double	Not Null	温度传感器 小数点后保留 1 位
湿度	hum	double	Not Null	湿度传感器 小数点后保留 1 位
光照	light	double	Not Null	光照传感器 小数点之后保留 1 位
上报时间	time	datetime	Not Null	采用数据库时间

9.4.3 CoAP API 设计

CoAP API 采用 REST 风格，每一个设备均被定义为一个资源，资源使用设备编号作为唯一标识，那么单个资源采用 /devices/<device_id> 这样的 URI 定义。请求负载可使用 JSON 格式，把温度传感器、湿度传感器、光照传感器结果填入 JSON 负载中。CoAP API 仅包括一个上传传感器检测结果的接口，接口的具体定义见表 9-2。

表 9-2 传感器数据上传 CoAP 接口

项　目	说　明
CoAP 请求类型	CON
CoAP 请求方法	POST
CoAP 请求路由	/devices/<device_id> ❑ device_id 表示设备编号
CoAP 请求媒体类型	application/json

CoAP 请求负载	{
	“temp”：<temp_value>,
	“hum”：<hum_value>,
	“light”：<light_value>
	}
	❑ temp 表示温度传感器结果
	❑ hum 表示湿度传感器结果
	❑ light 表示光照传感器结果

（续）

项　目	说　明
CoAP 响应类型	ACK
CoAP 响应码	❑ 2.01 Create 表示创建资源成功 ❑ 4.00 Bad Request 表示创建资源失败
CoAP 响应媒体类型	无
CoAP 响应负载	无

9.4.4　HTTP API 设计

HTTP API 设计包括页面路由和数据接口两部分，页面路由主要用于展现前端页面，并把从后端获得的数据展现至最终用户；而数据接口部分主要负责把数据库中的有效记录传输至前端页面。

1. 页面路由

（1）根页面

根页面也可称为主页面或默认页面，若用户访问该页面，服务器将渲染 admin.html。在微型物联网系统中只存在一个动态页面 admin.html，此时用户并没有选择具体设备，那么服务器将会根据预定义规则选择数据库中某台设备，如最新上传传感器数据的某个设备（见表 9-3）。

表 9-3　根页面

项　目	说　明
HTTP 请求方法	GET
HTTP 请求路由	/
HTTP 状态行	❑ 200 OK 表示成功获取页面 ❑ 404 Not Found 表示页面不存在
HTTP 响应媒体类型	html
HTTP 响应负载	使用最新的设备记录渲染 admin.html

（2）指定设备页面

与根页面不同，该页面路由中包括设备编号 deviceid，服务器将会根据设备编号渲染 admin.html（见表 9-4）。

表 9-4　指定设备页面

项　目	说　明
HTTP 请求方法	GET
HTTP 请求路由	/devices/<deviceid>/page ❑ deviceid 设备编号

（续）

项　目	说　明
HTTP 状态行	❑ 200 OK 表示成功获取页面 ❑ 404 Not Found 表示页面不存在
HTTP 响应媒体类型	html
HTTP 响应负载	使用指定设备记录，渲染 admin.html

2. 数据接口

在微型物联网系统中，数据接口可提供指定设备的历史记录。数据接口的路由为 /devices/<deviceid>/data.json，该路由和页面路由 /devices/<deviceid>/page 容易产生混淆，data.json 表示返回的结果为 JSON 格式，而 page 表示返回一个 HTML 页面。前端页面中表格使用 Bootstrap-table 组件，该组件通过异步的方式获取数据，获取数据时可设置查询条件，而服务器返回的数据也应符合 Bootstrap-table 的格式要求。

当请求数据时，一般数据库中的记录较多需要采用分页的方式分批获取，可通过 offset 参数和 limit 参数指定选择页码和每页记录数量，如每页显示 10 条记录，那么 limit 的值为 10，当获取第一页记录时 offset=0，当获取第二页数据时 offset=10，以此类推。

响应数据采用 JSON 格式，服务器返回的 JSON 对象中包含一个名为 total 的键值对，total 代表该设备传感器记录总数，JSON 对象中还包含一个名为 rows 的 JSON 数组，该 JSON 数组中的每一个元素对应一条传感器记录，而传感器记录又是一个 JSON 对象。该对象中包括表示温度传感器、湿度传感器和光照传感器的键值对。服务器将会根据请求中 limit 参数返回指定数量的传感器记录（见表 9-5）。

表 9-5　数据接口

项　目	说　明
HTTP 请求方法	GET
HTTP 请求路由	/devices/<deviceid>/data.json ❑ deviceid 设备编号
HTTP 请求参数	❑ offset 记录偏移量 ❑ limit 返回记录最大数量
HTTP 状态行	❑ 200 OK 表示成功获取页面 ❑ 404 Not Found 表示数据不存在
HTTP 响应媒体类型	application/json
HTTP 响应负载	{ "total": 12, "rows": [{ "id": 21, "device_id": "EE12",

（续）

项　目	说　明
HTTP 响应负载	"hum": 90, "temp": 24, "light": 182, "time": "2016-09-03 22:04:30" }, ……] } ❑ total：该设备传感器记录总数 ❑ rows：该设备传感器记录，使用 JSON 数组表示 ❑ id：记录序号 ❑ device_id：设备编号 ❑ hum：湿度结果 ❑ temp：温度结果 ❑ light：光照结果 ❑ time：传感器上传数据时间戳

9.5　具体实现

完成详细设计之后便可着手展开具体实现，微型物联网系统的实现代码可参考本书代码仓库 microsystem_server 文件夹，该文件夹中包含以下内容：

- ❑ app.js 主文件用于初始化 coap server 和 web server。
- ❑ package.json 微型物联网系统所有 Node.js 依赖组件，包括 express、swig、mysql node-coap 等。
- ❑ config 文件夹中包含 MySQL 连接信息文件 db-config.js。
- ❑ routes 文件夹中包含 coap_index.js 和 index.js 两个重要脚本文件，coap_index.js 包含 CoAP 服务的路由实现，index.js 包括 Web 服务的路由实现。
- ❑ public 文件夹中包含 Javascript、stylesheet 和 themes 三个文件夹，该文件夹主要保存和 Boostrap 有关的 CSS 样式和 Javascript 脚本。
- ❑ views 文件夹中包含 layout.html 和 admin.html，layout.html 是一个页面模板，而 admin.html 继承了该模板页面。
- ❑ lib 文件夹中包括 coap-router 组件，coap-router 是一个 node-coap 的中间件，可在 node-coap 中增加 express 框架类似的路由处理功能。
- ❑ test 文件夹包括用于测试 coap 路由的相关脚本，如用于模拟传感器数据上传的 post-sensor-data.js。

微型物联网系统的目录结构如图 9-5 所示。

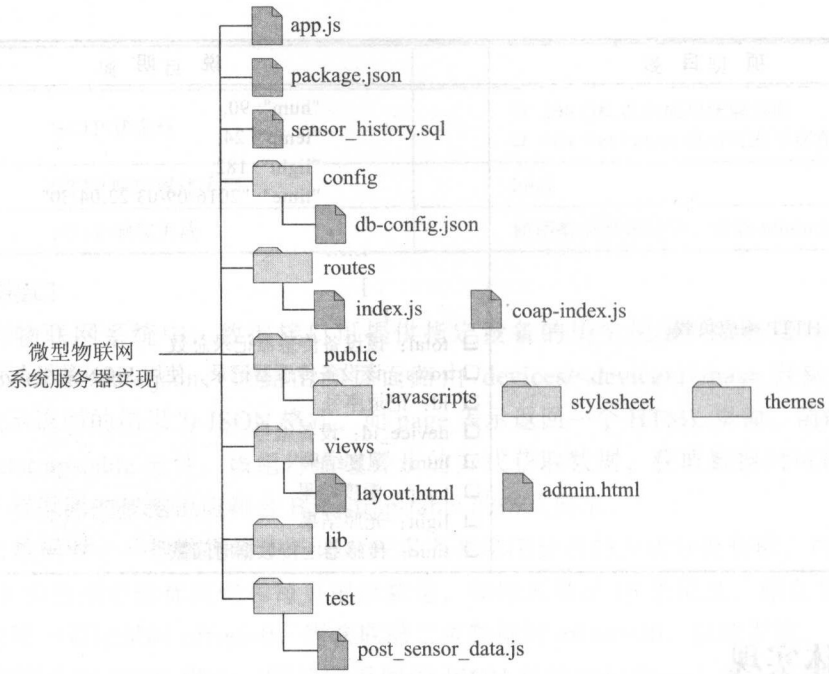


图 9-5　微型物联网系统目录结构

9.5.1　数据库实现

数据库是所有互联网或物联网应用的基础。先在树莓派中安装 MySQL 数据库，然后在数据库中创建表并在表中插入示例数据，最后进行一些简单的测试以保证数据库创建成功。

1. 安装 MySQL

树莓派中可使用 apt-get 安装 MySQL 数据库，在树莓派控制台中输入以下指令：

```
# 更新软件源
sudo apt-get update
# 安装MySQL数据库
sudo apt-get install mysql-server
```

一般情况下，MySQL 中存在一个默认用户“root”，在 MySQL 的安装过程中需要为 root 用户设置密码，设置密码的过程如图 9-6 所示。

2. 创建数据库和表

在 MySQL 中新建一个名为 monitor 的数据库，并在 monitor 数据库中创建名为 sensor_history 的数据表。

（1）登录 MySQL 控制台

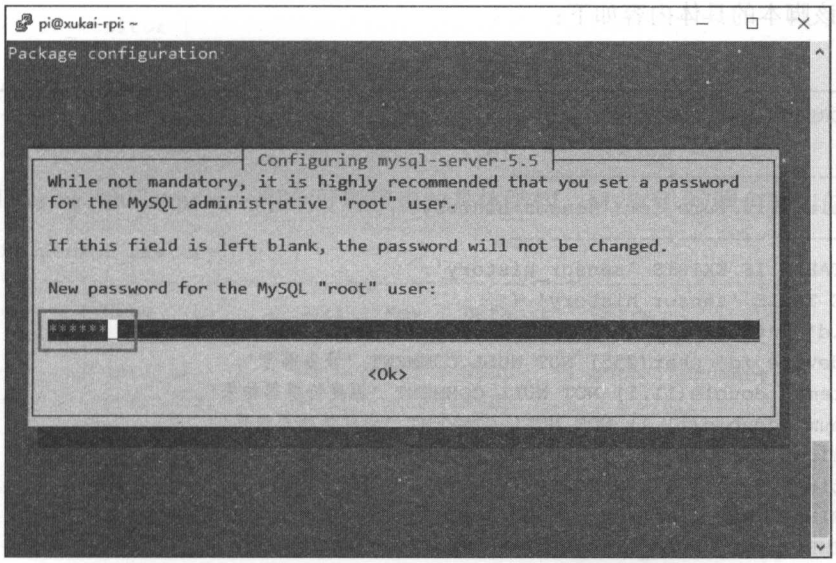


图 9-6　为 root 用户设置密码

```
# 使用root用户登录MySQL
mysql -u root -p
#输入root登录密码
Enter password:
```

使用 mysql 指令登录 MySQL 数据库，-u 参数用于指定用户，此处使用默认 root 用户，-p 代表用户登录密码，在控制台中输入“mysql –u root –p”，按下回车键之后控制台输出“Enter password”提示用户输入 root 用户登录密码。输入正确的登录密码便可顺利进入 MySQL 控制台。

（2）创建 monitor 数据库

在 MySQL 控制台中通过 create datebase 指令创建一个名为 monitor 的数据库，并切换到 monitor 数据库中。

```
mysql> create database monitor;
Query OK, 1 row affected (0.00 sec)
mysql> use monitor;
Database changed
```

- create database monitor：通过 create database 指令创建数据库，在 MySQL 控制台中使用分号作为命令结束符。
- use monitor：切换到 monitor 数据库。

（3）创建表和插入记录

在 microsystem_server 文件夹中包含一个 sensor_history.sql 脚本，该脚本实现了创建数据表和插入示例数据两个功能。在 mysql 控制台中通过 source 指令便可执行 sensor_history.

sql 脚本，该脚本的具体内容如下：

代码清单9-1 sensor_history.sql

```
SET FOREIGN_KEY_CHECKS=0;

-- ----------------------------
-- Table structure for 'sensor_history'
-- ----------------------------
DROP TABLE IF EXISTS 'sensor_history';
CREATE TABLE 'sensor_history' (
    'id' int(11) NOT NULL AUTO_INCREMENT,
    'device_id' char(255) NOT NULL COMMENT '设备编号',
    'temp' double(11,1) NOT NULL COMMENT '温度传感器结果',
    'hum' double(11,1) NOT NULL COMMENT '湿度传感器结果',
    'light' double(11,1) NOT NULL COMMENT '光照传感器结果',
    'time' datetime NOT NULL,
    PRIMARY KEY ('id')
) ENGINE=InnoDB AUTO_INCREMENT=4 DEFAULT CHARSET=utf8;

-- ----------------------------
-- Records of sensor_history
-- ----------------------------
INSERT INTO 'sensor_history' VALUES ('1', '12CD', '18.0', '58.0', '1802.0',
'2016-09-03 21:14:51');
INSERT INTO 'sensor_history' VALUES ('2', 'CD12', '28.0', '68.0', '2300.0',
'2016-09-03 21:16:49');
INSERT INTO 'sensor_history' VALUES ('3', 'EE12', '32.0', '70.0', '1900.0',
'2016-09-03 21:17:07');
```

使用 source 指令时需要指定 sensor_history.sql 脚本的绝对路径。本例中，sensor_history.sql 位于代码仓库 the_beginning_of_coap/microsystem_server 目录下。

```
source ~/repo/the_beginning_of_coap/microsystem_server/sensor_history.sql;
Query OK, 0 rows affected (0.00 sec)
Query OK, 0 rows affected (0.00 sec)
Query OK, 0 rows affected (0.02 sec)
Query OK, 1 row affected (0.01 sec)
Query OK, 1 row affected (0.01 sec)
Query OK, 1 row affected (0.01 sec)
```

（4）必要检查

完成表的创建之后需要进行一些必要的检查，以保证 sensor_history 表已经成功创建。使用 shows tables 指令可以查看 monitor 数据库中所有已经创建的表，MySQL 控制台输出结果如下：

```
show tables;
+-------------------+
```

```
| Tables_in_monitor |
+-------------------+
| sensor_history    |
+-------------------+
1 row in set (0.00 sec)
```

使用 describe <table_name> 指令可展现表的具体结构，MySQL 控制台输出结果如下：

```
describe sensor_history;
+-----------+-------------+------+-----+---------+----------------+
| Field     | Type        | Null | Key | Default | Extra          |
+-----------+-------------+------+-----+---------+----------------+
| id        | int(11)     | NO   | PRI | NULL    | auto_increment |
| device_id | char(255)   | NO   |     | NULL    |                |
| temp      | double(11,1)| NO   |     | NULL    |                |
| hum       | double(11,1)| NO   |     | NULL    |                |
| light     | double(11,1)| NO   |     | NULL    |                |
| time      | datetime    | NO   |     | NULL    |                |
+-----------+-------------+------+-----+---------+----------------+
6 rows in set (0.00 sec)
```

通过 select 查询语句可查看 sensor_history 表中已经被插入的记录，MySQL 控制台输出结果如下：

```
select * from sensor_history;
+----+-----------+------+------+-------+---------------------+
| id | device_id | temp | hum  | light | time                |
+----+-----------+------+------+-------+---------------------+
|  1 | 12CD      | 18.0 | 58.0 | 1802.0 | 2016-09-03 21:14:51 |
|  2 | CD12      | 28.0 | 68.0 | 2300.0 | 2016-09-03 21:16:49 |
|  3 | EE12      | 32.0 | 70.0 | 1900.0 | 2016-09-03 21:17:07 |
+----+-----------+------+------+-------+---------------------+
3 rows in set (0.00 sec)
```

到此，微型物联网系统数据库的部分已经正确完成。最后在控制台中输入“exit”以退出 MySQL 控制台。

9.5.2　CoAP 路由实现

CoAP 路由仅包括一个 /devices/</deviceid> 接口，该接口用于接收设备上传的传感器结果并将该内容插入 sensor_history 数据表中。

1. 启动 CoAP Server

在微型物联网系统中，app.js 可以理解为一个主脚本，该脚本用来初始化并启动 Web 服务和 CoAP 服务。为了更简单地说明 CoAP 服务的创建过程，首先新建一个名为 app-coap.js 脚本，该脚本中仅包括 coap 服务的具体实现代码，并没有 Web 服务相关代码。在 app-coap.js 中增加 CoAP 服务器实现，具体代码如下：

代码清单9-2 app-coap.js

```
// 模块依赖
var coap = require('coap');
// 引入coap路由
var coap_routes = require('./routes/coap_index');

var host = '0.0.0.0';
var coap_port = 5683;

// 启动coap服务器
coap.createServer(coap_routes).listen(coap_port);
console.log('CoAP Server Listening on :' + host + ':' + coap_port);
```

代码说明如下：

1）var coap = require('coap') 即载入 node-coap 模块。

2）var coap_routes = require('./routes/coap_index') 即 CoAP 路由的具体实现位于 coap_index.js 文件中。

3）coap.createServer(coap_routes).listen(coap_port) 即启动 CoAP 服务器并侦听 5683 端口，5683 端口为 CoAP 的默认服务端口。

2. 简单入手

在实现微型物联网系统指定的 CoAP 路由之前，先通过两个简单的示例说明 node-coap 如何与 MySQL“协同工作”。本小节示例包括两个路由：test 和 test-db，在具体项目中往往需要把复杂功能或未实践过的功能分成若干个“微小”的测试项目，通过一个又一个的简单测试来推进具体功能的实现。

（1）test 路由

在 coap_index.js 中增加一个 test 路由实现，具体代码如下：

```
// 测试目的路由
route.get('/test', function(req, res) {
    res.code = '2.05';
    res.end(new Date().toLocaleString());
});
```

test 路由在第 7 章也曾出现，该路由可返回服务器时间。app-coap.js 成功运行之后，可通过 Copper 插件检查 test 路由是否正确返回了服务器时间，具体结果如图 9-7 所示。

（2）test-db 路由

在 coap_index.js 中增加一个 test-db 路由，该路由可连接数据库，并执行一条 select 语句，test-db 路由的具体实现如下：

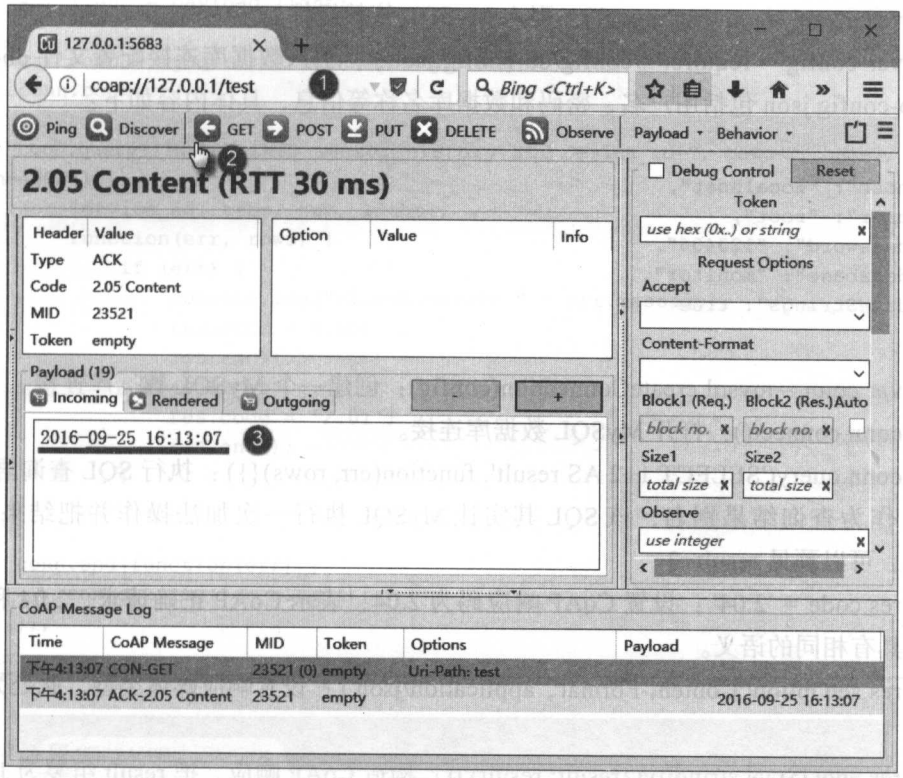


图 9-7　使用 Copper 访问 test 路由

代码清单9-3　test-db路由

```
var coap = require('coap');
var mysql = require('mysql');
var config = require('../config/db-config.json');

route.get('/test-db', function(req, res) {
    var conn = mysql.createConnection(config);
    conn.connect();

    conn.query('SELECT 1+2 AS result',
        function(err, rows) {
            var result = rows[0].result;
            res.code = '2.04';
            res.setOption('Content-Format', 'application/json');
            res.end(JSON.stringify({result: result}));
    });

    conn.end();
});
```

代码说明如下：

1）var mysql = require('mysql')：引入 MySQL 依赖组件。

2）var config = require('../config/db-config.json')：引入数据库连接配置文件 db-config.json，db-config.json 包括用户名、密码和数据库名称等信息，具体内容如下：

```
{
  "host": "localhost",
  "user": "root",
  "password": "123456",
  "database": "monitor",
  "dateStrings": true
}
```

3）var conn = mysql.createConnection(config)：创建一个 MySQL 数据库连接。

4）conn.connect()：打开 MySQL 数据库连接。

5）conn.query('SELECT 1+2 AS result', function(err, rows){})：执行 SQL 查询语句，使用 result 作为查询结果别名，该 SQL 其实让 MySQL 执行一次加法操作并把结果保存在 result 中，可以预见 result=3。

6）res.code = '2.04'：设置 CoAP 响应码为 2.04，表示 CoAP 正确请求，2.04 和 HTTP 200 OK 具有相同的语义。

7）res.setOption('Content-Format', 'application/json')：设置响应媒体类型，此处为 JSON 格式。

8）res.end(JSON.stringify({result: result}))：构造 CoAP 响应，把 result 组装为 JSON 格式并通过 res.end 反馈至 CoAP 客户端。

9）conn.end()：关闭 MySQL 数据库连接。

使用 Copper 插件访问 test-db 路由，访问 test-db 路由之后可以在 Incoming 选项卡中观察到 {"result": 3}，具体结果如图 9-8 所示。

3. 传感器路由实现

在 9.4.3 节中已经详细定义了传感器数据上传路由，传感器数据上传使用 CoAP POST 方法；请求 URI 中指定设备编号；请求负载中以 JSON 格式保存温度、湿度和光照结果。CoAP 服务器接收到该请求之后，把三种传感器结果插入 sensor_history 数据表中，具体实现代码如下：

代码清单9-4　传感器路由实现

```
route.post('/devices/{device_id}', function(req, res) {

    var device_id = req.params.device_id;
    console.log('device id: %s', device_id);

    var payload = JSON.parse(req.payload);
    var temp = payload.temp;
    var hum = payload.hum;
```

```
    var light = payload.light;

    var conn = mysql.createConnection(config);
    conn.connect();

    conn.query('INSERT INTO sensor_history SET device_id=?, temp=?, hum=?, light=?,
time=NOW()',
        [device_id, temp, hum, light],
        function(err, rows) {
            if (err) {
                console.log("client error: " + err.message);
                res.code = 4.00;
                res.end();
            } else {
                res.code = '2.01';
                res.end();
            }
    });

    conn.end(function(err) {
        console.log('db close');
    });
});
```

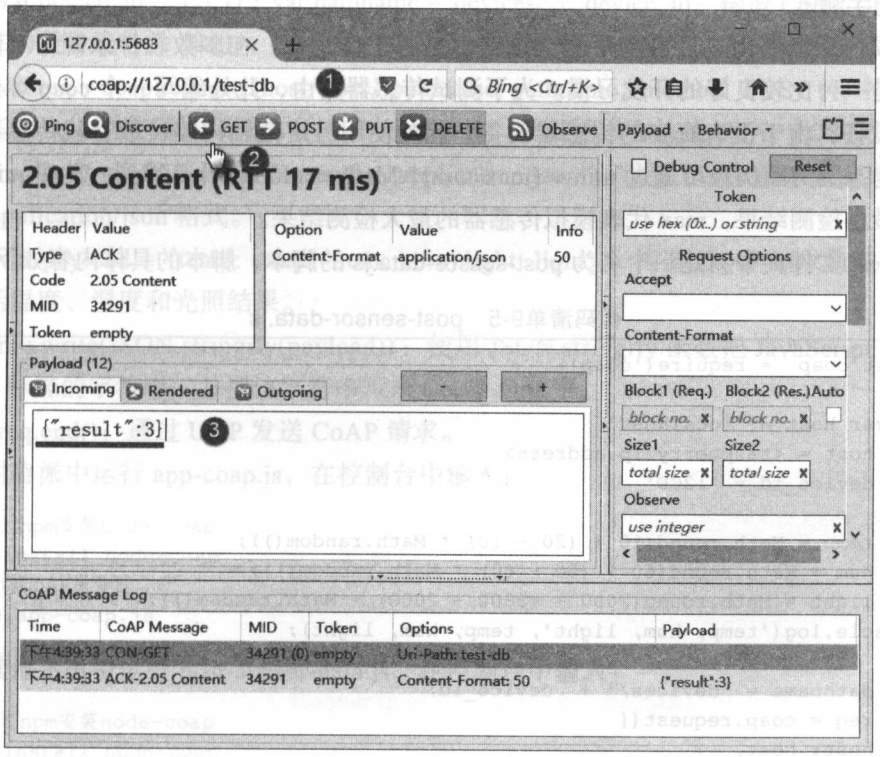


图 9-8　使用 Copper 访问 test-db 路由

代码说明如下：

1）route.post('/devices/{device_id}', function(req, res) {})：传感器数据上传路由，{device_ id} 表示传感器编号，传感器编号以字符串变量的形式存在于 URI 中，该 POST 路由将会处理所有以 /devices/ 开头的路由，如 /devices/12CD、/devices/EF12，其中 12CD 和 EF12 均为设备编号。

2）var device_id = req.params.device_id：获取请求 URI 中的 device_id 变量，该变量位于 req.params 对象中。

3）var payload = JSON.parse(req.payload)：通过 JSON.parse 把请求负载转化为 JSON 对象，通过 payload.hum 这样的方式便可获得湿度传感器结果。

4）INSERT INTO sensor_history SET device_id=?, temp=?, hum=?, light=?, time=NOW()：SQL 插入语句，把设备编号、传感器温度、传感器湿度、传感器光照和当前时间插入 sensor_history 数据表中。

5）conn.query('INSERT INTO SET device_id=?,..', [device_id, …], function(){})：在 SQL 插入语句中使用“?”作为占位符。

6）res.code = '2.01'; res.end()：若数据插入操作成功，CoAP 响应码设置为 2.01，表示资源创建成功，最后调用 res.end() 向 CoAP 客户端返回响应。

4. 动手测试

完成传感器路由之后可动手测试该路由是否工作正常，在多数软件系统中“做一步测一步”是一种比较良好的开发习惯。为了测试传感器路由，此处编写一个 coap 客户端模拟传感器设备，由于没有真实的传感器设备，此处使用一定范围内随机数替代传感器检测结果，一定范围的随机数可通过 min + (max-min)* Math.random() 计算得到，此时 min 模拟传感器的最小检测结果，max 代表模拟传感器的最大检测结果。

在 test 文件夹中新建一个名为 post-sensor-data.js 的脚本，脚本的具体内容如下：

代码清单9-5 post-sensor-data.js

```
const coap  = require('coap')

// var host = 'localhost'
var host = <raspberry ip address>
var device_id = '12CD'

var temp = Math.round(10 + (20 - 10) * Math.random());
var hum = Math.round(60 + (90 - 60) * Math.random());
var light = Math.round(2000 + (3000 - 2000) * Math.random());
console.log('temp, hum, light', temp, hum, light);

var pathname = 'devices/' +  device_id;
var req = coap.request({
    host: host,
    pathname: pathname,
```

```
    method: 'POST'
});
req.setOption("Content-Format", "application/json");

var payload = {
    temp: temp,
    hum: hum,
    light: light
}
req.write(JSON.stringify(payload));

req.on('response', function(res) {
    console.log('response code: ' + res.code)
    console.log('response payload: ' + res.payload)
})

req.end()
```

代码说明如下：

1）var temp = Math.round(10 + (20 - 10) * Math.random())：模拟一个温度传感器结果，温度传感器采用一个随机整数表示，最小结果为 10，最大结果为 20。湿度传感器和光照传感器采用相同的模拟方法。

2）var device_id = '12CD'; var pathname = 'devices/' + device_id：构造 CoAP URI-PATH，模拟请求的设备编号为 12CD。

3）var req = coap.request({host: host, pathname: pathname, method: 'POST'})：构造 CoAP 请求，其中 host 需使用树莓派的具体 IP 地址替换，此处请求方法为 CoAP POST 方法。

4）req.setOption("Content-Format", "application/json")：设置 CoAP 请求负载媒体类型，此处为 application/json 格式。

5）var payload = {temp: temp, hum: hum, light: light}：构造 CoAP 请求负载，请求负载中包括温度、湿度和光照结果。

6）req.write(JSON.stringify(payload))：使用 JSON.stringify 函数把 JavaScript 转换为符合 JSON 标准的字符串，并把该字符串写入 CoAP 负载中。

7）req.end()：通过 UDP 发送 CoAP 请求。

在树莓派中运行 app-coap.js，在控制台中输入：

```
# 通过npm安装node-coap
npm install node-coap
# 启动CoAP服务器
node app-coap.js
```

在其他主机中运行 post-sensor-data.js，在控制台中输入：

```
# 通过npm安装node-coap
npm install node-coap
```

```
# 启动CoAP客户端
node post-sensor-data.js
```

最后重新登录 MySQL 数据库，查看数据是否正确插入。

```
# 登录mysql
mysql -u root -p
Enter password:
Welcome to the MySQL monitor.
# 选择monitor数据库
mysql> use monitor;
# 查看sensor_history的所有记录
mysql> select * from sensor_history;
+----+-----------+------+------+--------+---------------------+
| id | device_id | temp | hum  | light  | time                |
+----+-----------+------+------+--------+---------------------+
|  1 | 12CD      | 18.0 | 58.0 | 1802.0 | 2016-09-03 21:14:51 |
|  2 | CD12      | 28.0 | 68.0 | 2300.0 | 2016-09-03 21:16:49 |
|  3 | EE12      | 32.0 | 70.0 | 1900.0 | 2016-09-03 21:17:07 |
|  4 | 12CD      | 14.0 | 79.0 | 2282.0 | 2016-10-02 12:08:40 |
+----+-----------+------+------+--------+---------------------+
4 rows in set (0.00 sec)
```

此时 sensor_history 表中增加了一条记录，温度传感器的模拟记录为 14，湿度传感器的模拟记录为 79，而光照传感器的模拟记录为 2282。通过该测试可以说明传感器路由已经被正确实现，一旦 CoAP 服务器接收到传感器数据，那么这些数据将被正确地插入 sensor_history 表中。

9.5.3 Web 前端实现

Web 前端主要由两个 HTML 文件组成：layout.html 和 admin.html，layout.html 是一个模板文件，该 HTML 中主要用于引入 CSS 样式和 JavaScript 脚本，该文件中并没有具体内容。admin.html 继承 layout.html，这样就不必重复引入各种 CSS 样式和 JavaScript 脚本，和 layout.html 不同，admin.html 将向用户展现动态数据。本例中 Web 前端使用 Bootstrap 框架，另外还使用基于 Bootstrap 的扩展组件——Bootstrap-table，该组件是一个简单实用的网页版“datagrid 控件”。

layout.html 和 admin.html 均位于 views 文件夹中，layout.html 和 admin.html 部分的编写工作除了与 CSS 样式、HTML 元素和 JavaScript 脚本有关之外，还与 Web 后台使用的模板渲染引擎相关。本例中 Web 后台使用 Express 框架，而 Express 框架的默认模板引擎为 jade，但本例使用了另一种模板引擎 swig，它和 Python Web 开发中常用的 Jinja 模板引擎非常相似。

1. layout.html

layout.html 具体实现代码如下：

代码清单9-6　layout.html

```
<!DOCTYPE html>
<html>
{% block head%}
<head>
    <meta charset="utf-8">
    <title>{% block title %}title{% endblock %}</title>

    {% block meta %}
     <meta name="viewport" content="width=device-width, initial-scale=1, maximum-
scale=1, user-scalable=no">
    {% endblock %}

    {% block styles %}
    <link rel="stylesheet" href="/stylesheet/custom.css">
    <link rel="stylesheet" href="/stylesheet/bootstrap.css">
    <link rel="stylesheet" href="/stylesheet/bootstrap-theme.css">
    <link rel="stylesheet" href="/stylesheet/bootstrap-table.min.css">
    {% endblock %}

    {% block scripts %}
    <script type="text/javascript" src="/javascripts/jquery.min.js"></script>
    <script type="text/javascript" src="/javascripts/bootstrap.min.js"></script>
     <script type="text/javascript" src="/javascripts/bootstrap-table.min.js"></
script>
    <script type="text/javascript" src="/javascripts/bootstrap-table-zh-CN.min.
        js"></script>
    {% endblock %}
</head>
{% endblock %}

<body>
    {% block content %}
    {% endblock %}
</body>
</html>
```

layout.html 部分说明如下：

1）{% block head%} {% endblock %}：使用 swig 模板中的 block 定义一个“块”，其他 html 文件若继承 layout.html 即可直接使用 block 定义的内容，也可以重新定义或追加 block 中所包含的内容。例如 layout.html 中定义“{% block title %}title{% endblock %}”，若 admin.html 没有重新定义 title block，那么 admin.html 的网页标题为“title”；若在 admin.html 定义“{% block title%} 微型物联网系统 {% endblock %}”，那么 admin.html 的网页标题为“微型物联网系统”。

2）layout.html 中还引入了 Bootstrap 框架相关的样式和主题，也引入了 Bootstrap-table 相关的样式和脚本。

3）admin.html 需要在 body block 中填入具体的内容。

2. admin.html

admin.html 主要由两部分组成，即左侧设备列表和右侧数据表格，在 Bootstrap 框架中页面宽度方向被分为 12 列，在本例中设备列表占据 2 列，使用 col-sm-2 样式定义；而右侧数据表格占 10 列，使用 col-sm-10 样式定义。在右侧数据表格中又可分为路径导航和 Bootstap-table 两部分，路径导航位于 Bootstrap-table 的上方。admin.html 最终外观如图 9-9 所示。

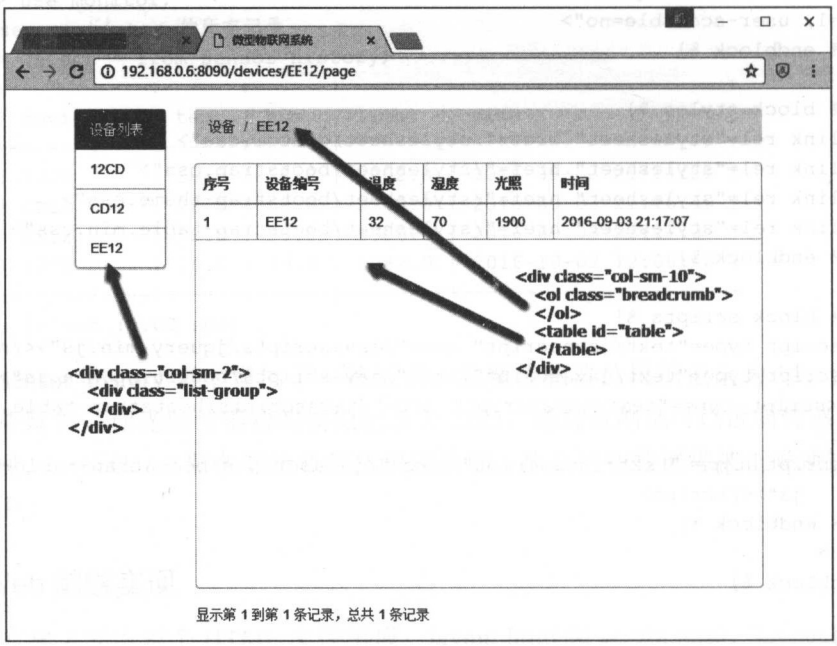


图 9-9　admin.html 外观

admin.html 的具体实现代码如下：

代码清单9-7　admin.html

```
{% extends 'layout.html' %}

{% block title%}
微型物联网系统
{% endblock %}

{% block content %}
<div class="container">

<div class="row">
    <div class="col-sm-2">
        <div class="list-group">
```

```
            <a href="#" class="list-group-item active">设备列表</a>
            {% for item in devices %}
            <a href="/devices/{{item.device_id}}/page" class="list-group-item">
            {{item.device_id}}
            </a>
            {% endfor %}
        </div>
    </div>
    <div class="col-sm-10">
    <ol class="breadcrumb">
        <li>设备</li>
        <li class="active">{{device_id}}</li>
    </ol>
        <table id="table"
            data-toggle="table"
            data-url="{{url}}"
            data-height="500"
            data-side-pagination="server"
            data-pagination="true"
            data-page-list="[5, 10, 20]"
            data-search="false"
            data-query-params="query_params">
        <thead>
            <tr>
                <th data-field="id" data-formatter="formatter_id">序号</th>
                <th data-field="device_id">设备编号</th>
                <th data-field="temp">温度</th>
                <th data-field="hum">湿度</th>
                <th data-field="light">光照</th>
                <th data-field="time">时间</th>
            </tr>
        </thead>
        </table>
    </div>
</div>
</div>

<script type="text/javascript">

function formatter_id(value, row, index) {
    return index + 1;
}

function query_params(params) {
    return {limit: params.limit, offset: params.offset}
}

$(function() {
    console.log('ready!');
```

```
});
</script>

{%   endblock %}
```

admin.html 代码解释如下：

1）{% extends 'layout.html' %}：admin.html 继承 layout.html 模板。

2）{% block title%} 微型物联网系统 {% endblock %}：重新定义网页标题，改为微型物联网系统。

3）使用 Bootstrap 中列表组显示设备列表，设备列表的具体数据由后台提供，后台向 swig 模板引擎传入一个名为 devices 的对象，devices 对象中使用数组的形式保存所有设备的 device_id，swig 模板引擎通过一次循环把 device_id “格式化”为 list-group-item 样式。在 swig 模板引擎中，if 和 for 这样的关键字使用 {% %} 标签包裹，如 {% for item in devices %}；而变量使用 {{ }} 包裹，如 {{item.device_id}}。

```
<div class="list-group">
    <a href="#" class="list-group-item active">设备列表</a>
    {% for item in devices %}
    <a href="/devices/{{item.device_id}}/page" class="list-group-item">
    {{item.device_id}}
    </a>
    {% endfor %}
</div>
```

4）路径导航可帮助用户获知当前查看的设备编号。

```
<ol class="breadcrumb">
    <li>设备</li>
    <li class="active">{{device_id}}</li>
</ol>
```

5）data-url="{{url}}"：设置 Bootstrap-table 从服务器获取数据的路径，数据路径由服务器渲染，变量名称为 {{url}}。

6）data-side-pagination="server"：Bootstrap-table 可使用前端分页或后端分页，此处设置为后端分页。

7）data-query-params="query_params"：由于 Bootstrap 采用后端分页，所以需要设置查询参数，Bootstrap-table 向后端请求数据前将运行 query_params 函数。

```
function query_params(params) {
    return {limit: params.limit, offset: params.offset}
}
```

Bootstrap 的默认查询参数有很多，但本例中仅使用了 limit 参数和 offset 参数，limit 参数表示单个分页包含多少个记录，而 offset 表示记录的偏移量，offset 参数和页码的含义非常类似。

8）定义表头，表头中 data-field 属性值应与数据库 sensor_history 表的字段名称保持严格一致，如 <th data-field="temp"> 温度 </th> 中，data-field="temp" 说明该列显示所有的温度传感器结果。

```
<tr>
    <th data-field="id" data-formatter="formatter_id">序号</th>
    <th data-field="device_id">设备编号</th>
    <th data-field="temp">温度</th>
    <th data-field="hum">湿度</th>
    <th data-field="light">光照</th>
    <th data-field="time">时间</th>
</tr>
```

9）Bootstrap-table 中还可以显示单元格实现内容，表格中的序号列并没有采用 sensor_history 中的 id 字段，而使用了表格中的行编号，Bootstrap-table 的行编号从 0 开始，而实际使用时行编号应从 1 开始。

```
function formatter_id(value, row, index) {
    return index + 1;
}
```

10）$(function() {console.log('ready!');}) ：用于检查 jQuery 是否载入成功。如果 jQuery 载入成功，可在浏览器控制台中观察到“ready！”。

9.5.4　Web 后端实现

Web 后端使用 Express 框架实现，Web 后端包含有两个功能。第一，根据用户的请求动态渲染 admin.html 页面；第二，浏览器载入 admin.html 后，Bootstrap-table 将根据 data-url 属性向后端请求表格数据，后端需要根据 Bootstrap-table 的预定义格式提供 JSON 数据。

1. 启动 HTTP Server

app.js 负责启动 HTTP 服务和 CoAP 服务，在 CoAP 路由实现小节中 app-coap.js 已经完成了启动 CoAP 服务的相关代码，app.js 需要在此基础上增加 HTTP 服务的实现代码。app.js 的具体实现代码如下：

代码清单9-8　app.js

```
// 模块依赖
var express = require('express')
var bodyParser = require('body-parser')
var path = require('path')
var swig = require('swig')
var coap = require('coap')

var routes = require('./routes/index');
var coap_routes = require('./routes/coap_index');
```

```
var app = express();

// 设置渲染引擎
app.engine('html', swig.renderFile);
app.set('view engine', 'html');

app.use(bodyParser.json());
app.use(express.static(path.join(__dirname, 'public')));

var coap_port = 5683;
var port = 8090;
var host = '0.0.0.0';

app.use('/', routes);
// 启动http服务器
app.listen(port, host);
console.log('Http Server Listening on :' +  host + ':' + port);

// 启动coap服务器
coap.createServer(coap_routes).listen(coap_port);
console.log('CoAP Server Listening on :' + host + ':' + coap_port);
```

1）var express = require('express')：增加 express 依赖项。

2）var routes = require('./routes/index')：引入所有的 Web 路由，这些路由在 routes 文件夹中的 index.js 脚本文件中实现。

3）app.engine('html', swig.renderFile)app.set('view engine', 'html')：设置渲染引擎，本例采用 swig 模板引擎，所有扩展名为 html 的文件将使用 swig 模板引擎进行渲染。

4）app.use(express.static(path.join(__dirname, 'public'))) ：指定相关静态文件路径，这些文件包括 CSS 样式和 JavaScript 脚本，这些文件都保存在 public 文件夹中。此处相关 CSS 样式和 JavaScript 脚本的根目录为 public，其路径总是相对于 public 文件夹，如 <script src="/javascripts/jquery.min.js"></script>。

5）app.use('/', routes) Express 框架接收到的所有用户请求均由 routes 进行处理，routes 指向 routes 文件夹内的 index.js 文件。

6）app.listen(port, host)：在指定端口启动 HTTP 服务。

2. 设备路由

设备路由需要实现两部分功能：第一，若用户没有选择设备编号，那么服务器返回用户默认设备的历史数据；第二，若用户选择单个设备，那么服务器返回具体设备的历史数据。在 /routes/index.js 文件中增加一个获取所有设备编号的函数——get_device_list:

```
function get_device_list(callback) {
    var conn = mysql.createConnection(config);
    conn.connect();
```

```
    conn.query("SELECT DISTINCT device_id FROM sensor_history",
        function(err, rows) {
            callback(rows);
    });

    conn.end();
}
```

get_device_list 函数的具体说明如下：

1）SELECT DISTINCT device_id FROM sensor_history：使用 DISTINCT 关键字查询所有不重复的 device_id，该条 SQL 语句将返回 sensor_history 中所有不重复的设备编号。

2）callback(rows)：通过回调的方式返回所有记录。

设备路由可处理两类路由，即“/”和“/devices/:device_id/page”，在 index.js 中通过以下代码实现路由处理。

```
router.get('/', function (req, res) {
    get_device_list(function(rows){
        var device_id = rows[0].device_id
        var url = '/devices/' + device_id + '/data.json';
        res.render('admin.html', {url: url, device_id: device_id, devices: rows});
    })
});

router.get('/devices/:device_id/page', function (req, res) {
    var device_id = req.params.device_id;
    var url = '/devices/' + device_id +'/data.json';
    get_device_list(function(rows){
        res.render('admin.html', {url: url, device_id: device_id, devices: rows});
    })
});
```

这两个路由非常相似，处理“/”路由时，device_id 采用查询结果的第一条记录；而处理“/devices/:device_id/page”时，device_id 已经由用户在 URL 中定义。具体代码说明如下：

1）get_device_list(function(rows){...})：调用 get_device_list 获取所有不重复的设备记录，在调用 get_device_list 通过匿名回调函数获取结果。

2）var url = '/devices/' + device_id + '/data.json'：向前端 Bootstrap-table 传入获取表格数据的 URL，如“device_id”为 CD12，那么获取表格数据的 URL 为“/devices/CD12/data.json”。

3）res.render('admin.html', {url: url, device_id: device_id, devices: rows})：使用 url、device_id 和 devices 参数渲染 admin.html，其中 url 和 device_id 可理解为单个变量，而 devices 为设备编号的集合包含多个结果。

3. 传感器数据接口

传感器数据接口是一个返回 JSON 格式数据的异步接口，该接口专门为 Bootstrap-table 服务。

```
router.get('/devices/:device_id/data.json', function (req, res) {
    var device_id = req.params.device_id;
    var offset = parseInt(req.query.offset) || 0;
    var limit = parseInt(req.query.limit) || 10;
    var total = 0;
    var rows = null;

    var conn = mysql.createConnection(config);
    conn.connect();

    conn.query('SELECT COUNT(*) as total FROM sensor_history WHERE device_id=?',
        [device_id],
        function(err, result) {
            total = result[0].total;
    });

     conn.query('SELECT * FROM sensor_history WHERE device_id=? order by id desc
limit ?, ?',
        [device_id, offset, limit],
        function(err, result) {
                rows = result;
    });

    conn.end(function(err) {
        res.json({total:total, rows:rows});
    });
});
```

该部分代码说明如下：

1）var device_id = req.params.device_id：通过 req.params 获取“URL”中的 device_id 变量。

2）var offset = parseInt(req.query.offset) || 0; var limit = parseInt(req.query.limit) || 10：通过 req.query 获取请求参数 offset 和 limit，其中 offset 的默认值为 0，而 limit 的默认值为 10。

3）SELECT COUNT(*) as total FROM sensor_history WHERE device_id=?：通过 COUNT (*) 和 WHERE 关键字查询指定 device_id 的传感器历史数据记录总数。

4）SELECT * FROM sensor_history WHERE device_id=? order by id desc limit ?, ?：使用 limit 关键字限制查询结果的数量，通过 JSON 数组的方式向查询语句传递三个参数——device_id、offset、limit，其中 offset 参数可指定查询结果初始位置，而 limit 参数可指定查询结果数量。

5）res.json({total:total, rows:rows})：当两次查询动作完成之后通过 res.json 返回 JSON 结果。res.json 是一个返回 JSON 媒体类型的简便方法，该函数将自动填充 HTTP 首部并把 JavaScript 对象转化为 JSON 字符串。

4. 动手测试

完成设备路由和传感器数据接口之后，可进行一些简单的测试以保证这些接口工作正常。在 9.5.1 节，创建 sensor_history 表时已经在表中插入了三条记录，三条记录分别属于不同的设备编号——12CD、CD12 和 EF12，那么访问 /devices/12CD/data.json 至少可以获得一条记录。

在树莓派控制台中运行 app.js：

```
# 运行所有服务器
node app.js
# 控制台打印
Http Server Listening on :0.0.0.0:8090
CoAP Server Listening on :0.0.0.0:5683
```

在浏览器地址栏内输入 http://<raspberry_ip>:8090/devices/12CD/data.json，若传感器数据接口工作正常，可在浏览器中可观察到如图 9-10 所示输出。

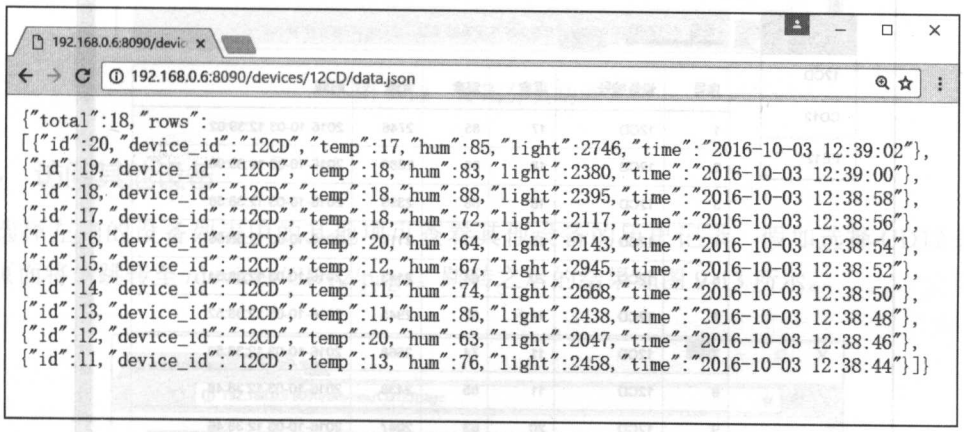


图 9-10　访问传感器数据接口

9.6　综合测试

下面我们为微型物联网系统做一轮综合测试，通过综合测试验证功能是否符合设计需求。

9.6.1　启动微型物联网系统

在树莓派启动 Web 服务器和 CoAP 服务器，在控制台中输入以下指令：

```
# 启动Web服务器和CoAP服务器
node app.js
# 控制台输出
```

```
Http Server Listening on :0.0.0.0:8090
CoAP Server Listening on :0.0.0.0:5683
```

9.6.2 增加模拟数据

为了更好地测试微型物联网系统，可以通过 test 文件夹中的 post_sensor_data.js 脚本向 CoAP 服务器插入更多的模拟数据。

9.6.3 访问默认设备

在浏览器地址栏中输入树莓派的 IP 地址和服务端口号，此处树莓派的 IP 地址为 192.168.0.6，服务端口号为 8090，在浏览器中输入“192.168.0.6:8090”，具体结果如图 9-11 所示。

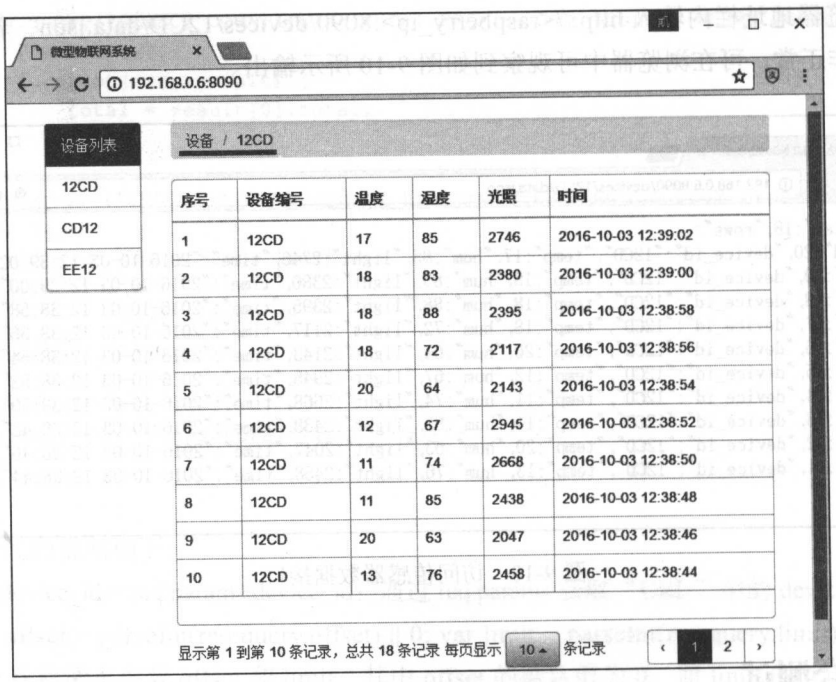


图 9-11 访问默认设备

虽然在 URL 中并没有指定设备编号，但是 HTTP 服务器依然返回了默认设备 12CD 的部分历史记录。

9.6.4 使用分页功能

由于 12CD 设备的传感器记录较多，所以 Bootstrap-table 采用分页显示的方式展现历史记录，默认情况下每页显示 10 条记录，用户可选择网页右下角的页码查看其他历史记录，分页查询结果如图 9-12 所示。

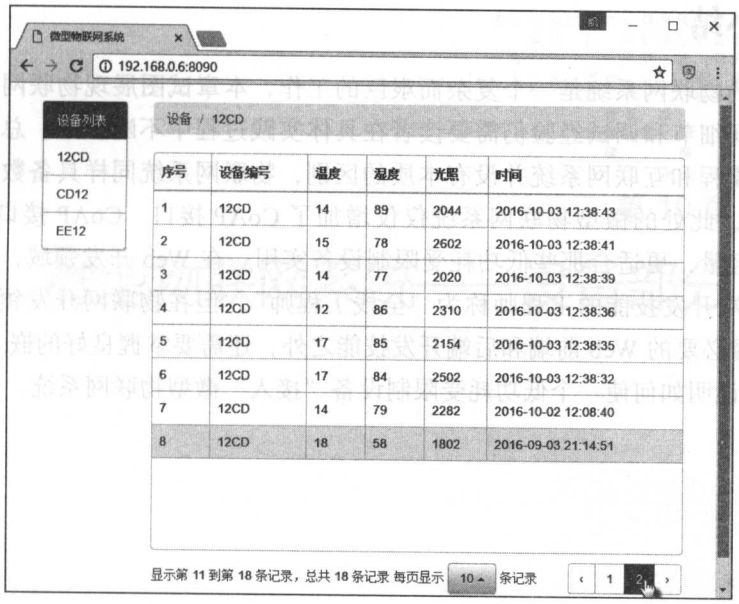


图 9-12　使用分页功能

9.6.5　访问其他设备

选择左侧的设备列表中的其他项可查看其他设备的历史记录，假如选择 CD12 设备，那么页面将会跳转至 /devices/CD12/page。跳转之后的结果如图 9-13 所示。

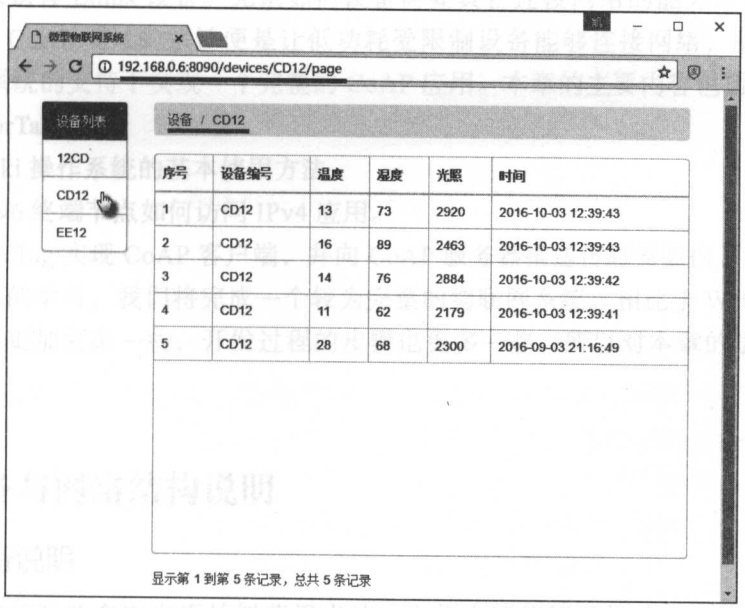
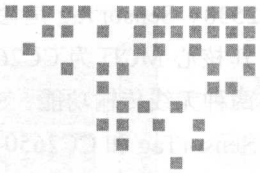


图 9-13　访问其他设备

9.7 本章小结

开发完整的物联网系统是一个复杂而艰巨的工作，本章试图展现物联网系统的开发流程，更多的实现细节和调试经验仍需要读者在具体实践过程中不断积累。总的来说，物联网系统的开发流程和互联网系统并没有本质的区别，物联网系统同样具备数据库、Web 前端和 Web 后端，此处的微型物联网系统仅仅增加了 CoAP 接口。CoAP 接口相比于 HTTP 接口显得较为轻量，更适合那些低功耗受限制设备实用。在 Web 开发领域，同时掌握 Web 前端和 Web 后端开发技能的工程师称为“全栈工程师”。但在物联网开发领域，物联网工程师除了要掌握必要的 Web 前端和后端开发技能之外，还需要掌握良好的嵌入式开发技能，第 10 章将详细说明如何使一个低功耗受限制设备“接入”微型物联网系统。



第 10 章 Chapter 10

微型物联网系统——设备部分

10.1 本章主要内容

微型物联网系统不仅包括服务器部分的具体实现，也包括设备部分的具体实现。第 9 章我们已经完成了服务器部分的开发工作，本章我们将继续推进设备部分的开发工作。在物联网系统中，终端设备的开发方式多种多样，可采用 Arduino 这样的小型 MCU 设备，也可使用树莓派这样 Linux 设备。无论如何设备需要具备连接网络的能力才可以实现 CoAP 客户端功能。CoAP 设计的宗旨便是让低功耗受限制设备能够连接网络，所以本章试图在 Contiki 操作系统的支持下实现一个完整的 CoAP 应用。本章的主要内容包括：

- SensorTag 简介。
- Contiki 操作系统的基本使用方法。
- 纯 IPv6 终端节点如何访问 IPv4 应用。
- SensorTag 实现 CoAP 客户端，并向 CoAP 服务器推送传感器数据。

经过本章的学习，我们将完成一个较为完整的物联网系统。相比于 Web 开发，嵌入式设备的开发要更加复杂一些，开发过程的步骤也更多一些，所以对本章的学习需要很大的耐心。

10.2 设备与网络结构说明

10.2.1 设备说明

除了在其他章节多次出现的树莓派之外，本例中还使用了另外两个设备：SensorTag/

CC2650 和 CC2538。SensorTag 是德州仪器推出的 CC2650 开发套件，该套件具有多种不同的传感器，其核心 MCU 为 CC2650。而 CC2650 是新一代无线 SoC，同时具有 BLE 和 IEEE 802.15.4 两种无线传输功能，SensorTag 是本例的“主角”。

更多关于 SensorTag 和 CC2650 的内容请参考德州仪器官方网站：

- SensorTag：http://processors.wiki.ti.com/index.php/CC2650_SensorTag_User's_Guide
- CC2650：http://www.ti.com.cn/product/cn/CC2650

除了“主角”SensorTag 之外，本例还包括它的“好帮手”CC2538。CC2538 是一款符合 IEEE 802.15.4 标准的无线 SoC，虽然与 CC2650 型号不同，但在 2.4G 频段中两者可以进行正常的无线通信。CC2538DK 是德州仪器推出的 CC2538 开发套件之一，但是随着 CC2538 的流行也可以在其他平台购买到与 CC2538DK 兼容的 CC2538 开发套件。

更多关于 CC2538 和 CC2538DK 的内容请参考德州仪器官方网站：

- CC2538：http://www.ti.com.cn/product/cn/CC2538
- CC2538DK：http://www.ti.com.cn/tool/cn/cc2538dk

SensorTag 与 CC2538 开发板的实际外观如图 10-1 所示，CC2538 开发板与德州仪器推出的 CC2538DK 保持兼容。

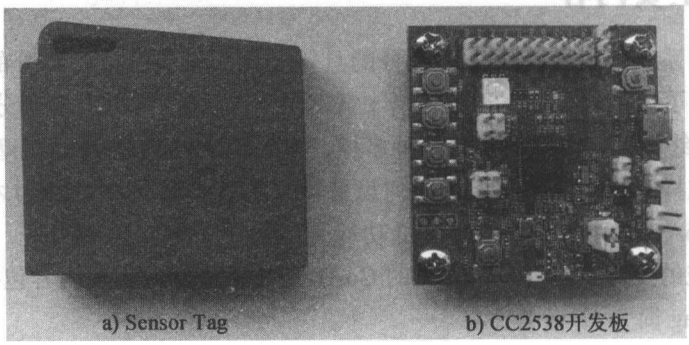


图 10-1　SensorTag 与 CC2538 外观

1. SensorTag/CC2650

本章示例中并没有使用 SensorTag 的所有功能，而仅使用了 SensorTag 中的温湿度传感器 HDC1000、光照传感器 OPT3001 和 UART、LED、按键等基本功能。其中 SensorTag 的 UART 功能主要用于调试，LED 将指示 SensorTag 工作状态，按键功能将触发 SensorTag 执行具体工作，SensorTag 的交互能力非常有限，但是对于一个物联网设备来说这些已经足够了。关于 HDC1000 和 OPT3001 这两个传感器的更多内容请参考德州仪器官方网站：

- HDC1000：http://www.ti.com.cn/product/cn/HDC1000
- OPT3001：http://www.ti.com.cn/product/cn/OPT3001

SensorTag/CC2650 的结构示意图如图 10-2 所示。

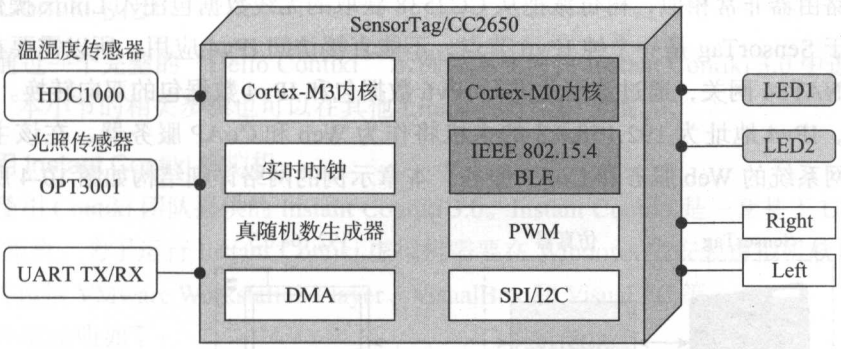


图 10-2　SensorTag/CC2650 结构示意图

2. CC2538 开发板

与 SensorTag 相似，CC2538 开发板也具有多种功能，但是本章示例中 CC2538 仅作为 Slip-Radio 使用，也就是说本例仅使用了 CC2538 的无线功能和串口功能，CC2538 将作为 SensorTag 连接传统网络的重要“桥梁”。CC2538 开发板的结构示意图如图 10-3 所示。

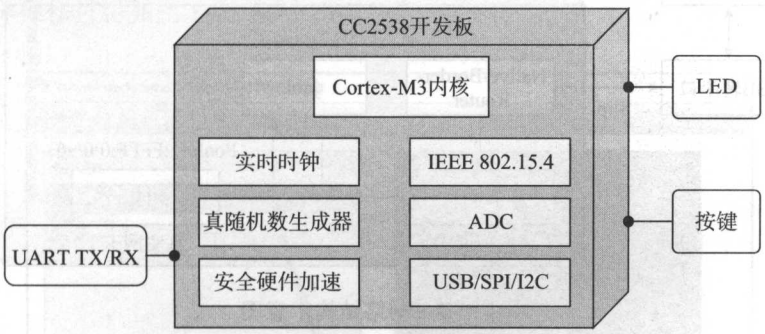


图 10-3　CC2538DK 结构示意图

10.2.2　网络结构说明

本章的示例中至少包括三个主要设备：SensorTag、CC2538 和树莓派。其中 SensorTag 作为 CoAP 客户端，SensorTag 与专用下载器 XDS110 相连，XDS110 不但可以下载固件至 SensorTag 中，还可以与 SensorTag/CC2650 的串口相连，以便在 PC 中虚拟一个串口设备，通过该串口设备可以方便地查看 SensorTag/CC2650 的运行情况；另外 CC2538 将作为 Slip-Radio，把经过 6LoWPAN 压缩的 IPv6 数据包变为 Slip 串行数据并通过自身串口传输至树莓派中，由于并没有直接使用树莓派中的物理串口，所以 CC2538 通过一个 USB 转串口设备插入树莓派 USB 端口中，对树莓派来说 CC2538 是一个名为 ttyUSB0 的串行设备。

本例中树莓派作为边界路由。在 6LoWPAN/IPv6 低功耗网络中边界路由的职责与 WiFi

网络中的路由器非常相似，树莓派把从 CC2538 获取的无线数据包注入 Linux 操作系统的网络层，由于 SensorTag 是一个纯 IPv6 节点，不能直接访问 IPv4 应用，所以需要在树莓派中增加一个 NAT64 网关，通过该网关进行 IPv6 数据包和 IPv4 数据包的双向转换。

最后，IPv4 地址为 192.168.0.3 的主机将作为 Web 和 CoAP 服务器，在该主机中运行微型物联网系统的 Web 服务和 CoAP 服务。本章示例的网络详细结构如图 10-4 所示。

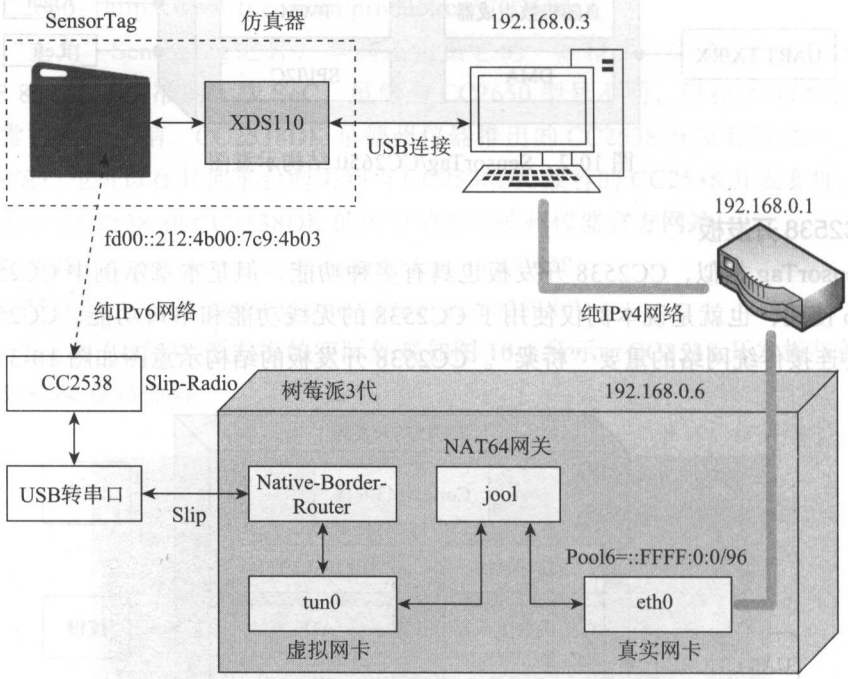


图 10-4　网络结构示意图

10.3　Contiki 入门

与其他嵌入式开发不同，Contiki 所支持平台的开发工作一般在 Linux 下进行，常用的 Linux 发行版如 Ubuntu、Debian 和 Deepin 等均可进行 Contiki 开发。为了简化 Contiki 的学习门槛，Contiki 开发团队提供了一套集成开发环境——Instant Contiki。Instant Contiki 是一套已经安装好各种依赖包和工具链的 Ubuntu 14.04 虚拟机，不需要任何配置便可进行 Contiki 开发。在 Instant Contiki 中已经预装了 ARM-Cortex M3 和 MSP430 的交叉工具链。Instant Contiki 的更多信息请参考 Contiki 官方网站 www.contiki-os.org。

除了 Instant Contiki 集成开发环境之外，其他 Linux 发行版也可以进行 Contiki 开发。在这种情况下用户需要手动安装各种依赖工具和交叉工具链，依赖工具包括 curl、git、wget 和 gcc 等，交叉工具链包括 gcc-arm-none-eabi 和 gcc-msp430 等。

10.3.1　Contiki 初步

下面通过一个完整的“Hello Contiki”示例说明如何在 Instant Contiki 3.0 中进行 Contiki 开发工作。本小节的相关步骤也可以在其他 Linux 发行版中实现。

1. 使用 Instant Contiki 虚拟机

本节使用 Contiki 团队提供的 Instant Contiki 3.0。Instant Contiki 是一套基于 Ubuntu 14.04 的 32 位虚拟机，为了运行 Instant Contiki 虚拟机需要在 Windows 中安装虚拟机软件，常见的虚拟机软件包括 VMware Workstation Player、VisualBox 和 Visual PC 等。

实验环境说明如下：

- 宿主机：Windows 10
- 虚拟机软件：VMware Workstation Player 12.0
- Linux 发行版：Instant Contiki 3.0(Ubuntu 14.04 LTS)

打开 VMware Workstation 12 Player 之后，在开始界面的左侧选择 Instant Contiki 3.0，在开始界面的右下角选择“播放虚拟机”。进入 Instant Contiki 之后需要输入 user 用户的登录密码，默认情况下 Instant Contiki 的默认用户为 user，登录密码为 user。进入 Instant Contiki 的具体操作过程如图 10-5 和图 10-6 所示。

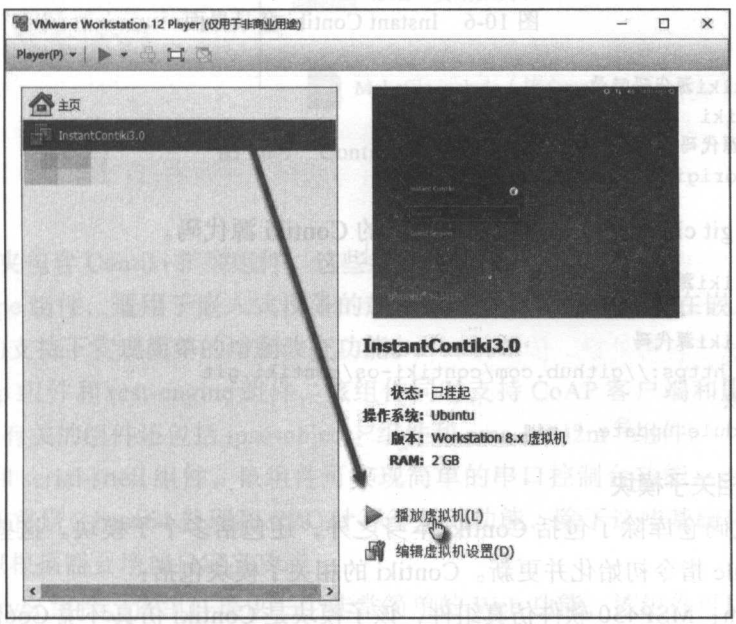


图 10-5　选择 Instant Contiki 虚拟机

2. 获取 Contiki 源代码

在 Instant Contiki 3.0 中已经预装了 Contiki 的源代码，但该部分代码并不是最新代码，

Contiki 源代码位于 user 用户目录下的 contiki 文件夹中。若需要与 Contiki 代码仓库 master 分支同步，可进入 Contiki 目录后输入 git pull origin 以获取最新代码，具体操作如下：

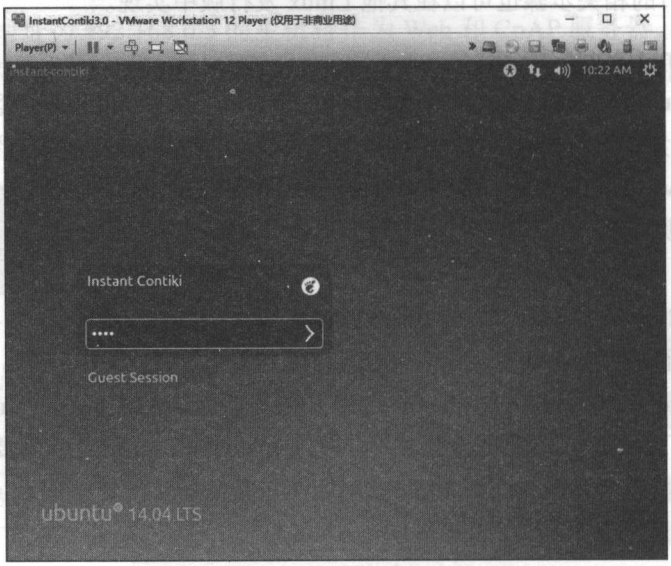


图 10-6　Instant Contiki 登录界面

```
# 进入Contiki源代码目录
cd ~/contiki
# 拉取最新源代码
git pull origin
```

另外使用 git clone 指令也可以获取最新的 Contiki 源代码。

```
# 复制Contiki源代码到用户目录
cd ~
# 获取Contiki源代码
git clone https://github.com/contiki-os/contiki.git
# 获取子模块
git submodule update -init
```

3.Contiki 相关子模块

Contiki 代码仓库除了包括 Contiki 本身之外，还包括多个子模块。这些子模块需要使用 git submodule 指令初始化并更新。Contiki 的相关子模块包括：

- mspsim：MSP430 软件仿真组件，该子模块是 Contiki 仿真环境 Cooja 的必要组件。
- cc2538-bsl：CC2538 平台串口 bootload 工具，该工具使用 Python 开发。cc2538-bsl 除了适用于 CC2538 平台之外，也适用 CC2630 和 CC1310 等 SoC。
- cc26xxware/cc13xx ware ：CC2650 和 CC1310 外设驱动库。该子模块是 CC2650 和 CC1310 开发的必要组件。

4. Contiki 源代码结构

Contiki 源代码包含多个目录，这些目录包括 apps（应用组件）、core（核心代码）、cpu（移植与驱动）、platform（平台实现）、examples（基础示例）和 tools（实用工具）等。Contiki 的源代码结构如图 10-7 所示。

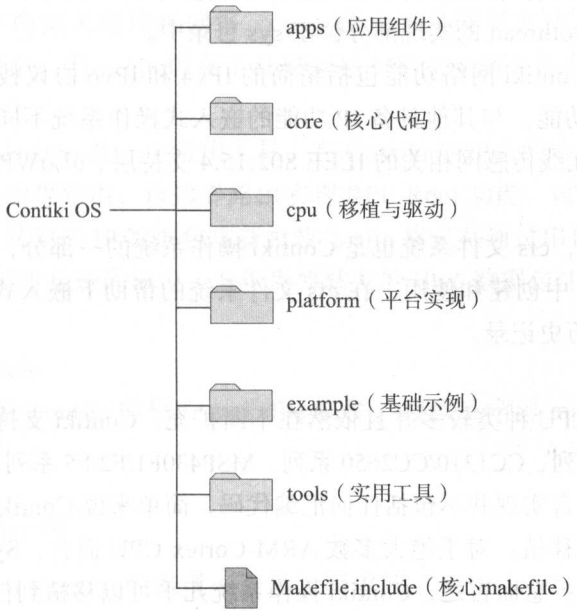


图 10-7　Contiki 源代码结构

（1）apps

apps 文件夹包含 Contiki 扩展组件，这些组件包括：

- antelope 组件，适用于嵌入式设备的超精简型数据库系统，可在嵌入式 Flash 文件系统的支持下实现简单的增删改查功能。
- er-coap 组件和 rest-engine 组件，该组件同时支持 CoAP 客户端和服务器功能，与 CoAP 有关的组件还包括 ipso-object[⊖]组件和 oma-lwm2m[⊜]组件。
- shell 和 serial-shell 组件，该组件可实现简单的串口控制台功能，shell 组件提供诸如设备重启、base64 处理和 CRC 计算等基础功能，除了这些基础功能之外，用户还可以根据需要增加自定义功能。
- webserver 组件，该组件提供一个非常简单的 Web 功能，该组件可以帮助设备实现一些简单的预定义网页。

⊖ https://www.ipso-alliance.org/。

⊜ http://openmobilealliance.org/iot/。

（2）core

core 文件夹包含 Contiki 的所有核心代码，这些核心功能代码包括：

- protothread 任务调度功能，Contiki 采用事件触发机制，其核心为 protothread 任务调度机制，该机制和多数抢占式的操作系统存在明显区别，在 protothread 机制下任务没有私有堆栈，任务进行切换时操作系统仅保留 C 文件中的行号作为上下文切换的依据，protothread 的实现部分位于 sys 目录中。
- 网络功能，Contiki 网络功能包括精简的 IPv4 和 IPv6 协议栈，支持 UDP、TCP 和部分 HTTP 功能，与其他具备 IP 功能的嵌入式操作系统不同，Contiki 网络部分还包括低功耗无线传感网相关的 IEEE 802.15.4 支持层、6LoWPAN 头压缩层和 RPL[⊖] 路由功能等；
- cfs 文件系统，cfs 文件系统也是 Contiki 操作系统的一部分，该文件系统可在 CPU 内部的 Flash 中创建和使用，在 cfs 文件系统的帮助下嵌入式设备可以方便地保存数据或读取历史记录。

（3）cpu

Contiki 支持的 CPU 种类较多并且依然在不断扩充，Contiki 支持的 CPU 包括 STM32F 152 系列、CC2538 系列、CC1310/CC2650 系列、MSP430F1/F2/F5 系列、nRF52 系列。Contiki 操作系统完全由 C 语言实现并不包括任何汇编代码，简单来说 Contiki 操作系统仅需实现一个系统时钟便可完成移植，对于绝大多数 ARM-Cortex CPU 而言，Systick 时钟是一个非常合适的系统时间载体。总而言之，Contiki 操作系统几乎可以移植到任何 CPU 中。cpu 目录中有一个特殊的 cpu——native，该 cpu 可以理解为把 Contiki 操作系统“移植”到 Linux 系统中，那么 Contiki 的部分功能也可以在 Linux 平台下实现并验证。

（4）platform

platform 文件夹包含 Contiki 操作系统所支持的各类平台，这些平台包括 CC2538DK 平台、SRF06-CC26xx(CC1310/CC2650）平台，STM32 NUCLEO 平台等。在 CPU 移植的基础上，这些平台还增加了各种外设、传感器和执行器。每个平台下都包含 contiki-main.c 和 contiki-conf.h，contiki-main.c 中包含 main 函数，实现了绝大多数初始化工作，如操作系统初始化、网络组件初始化、辅助功能初始化和低功耗实现等功能，在 platform 的帮助下用户仅需要关心具体功能实现便可；contiki-conf.h 包含与平台相关的默认 Contiki 配置，熟悉 Contiki 的所有配置并不是一件容易的事情。用户也可以参考相似的平台创建符合自身要求的 platform。

（5）examples

examples 文件夹包括大量基础示例，这些示例包括不同平台下的 UDP 客户端服务器示例、传感器示例、任务调度示例、定时器组件（etimer、ctimer 和 rtimer）示例；除了具体平台的示例之外，examples 平台还包括边界路由示例、Slip-Radio 示例等实用功能。

⊖ https://datatracker.ietf.org/doc/rfc6550/。

（6）tools

tools 文件夹中包含很多使用 Contiki 过程中的实用工具，这些实用工具包括：

- CC2538-bsl 工具：使用该工具可以通过串口下载固件，CC2538-bsl 支持 CC2538、CC2650 和 CC1310 等。
- cooja 仿真工具：cooja 仿真工具是 Contiki 的一个特色工具，借助该工具可以在纯软件环境下模拟无线传感网络，cooja 仿真工具既可支持多个终端节点，也可以支持边界路由，但 cooja 仿真工具仅支持 MSP430 相关平台。并不能仿真 ARM Cortex 系列平台。
- tools 目录中 tunslip6 是一个常用工具，在 Contiki 应用系统中低功耗无线传感网络一般存在一个边界路由，该边界路由实现 RPL Root 功能，该路由设备将把低功耗传感网中需要转发的 IP 数据包重新组装为 slip 格式并通过串口发送至 Linux 主机，tunslip6 工具侦听指定的串口，并把需要转发的 IPv6 数据包注入到 Linux 网络层驱动中。

（7）makefile.include

makefile.include 是 Contiki 的核心 makefile 文件，阅读或学习 makefile.include 的细节可不是一件容易的事情。

10.3.2　native 入门示例

下面我们在 Instant Contiki 中实现一个最为简单的“hello world”示例，该示例位于 examples/hello-world 目录中，通过 cd 指令进入该目录后运行“make TARGET=native”便可生成可执行文件，具体操作步骤如下：

```
# 进入hello-world目录
cd ~/contiki/exapmples/hello-world
# 使用native平台
make TARGET=native
```

TARGET 参数用于指定平台，native 平台是 Contiki 所支持的平台中较为特殊的平台，native 平台一般特指 Linux 平台，该平台在 Linux 系统内仿真 Contiki 系统所需的运行环境。此时经 make 编译获得的可执行文件为 hello-world.native，该文件可在 Linux 系统中直接运行。在控制台中输入“./hello-world.native”可获得类似输出。

```
# 运行可执行文件./hello-world.native
./hello-world.native
# contiki-main.c中输出
Contiki-3.x-2906-g14bfaff started with IPV6, RPL
Rime started with address 1.2.3.4.5.6.7.8
MAC nullmac RDC nullrdc NETWORK sicslowpan
Tentative link-local IPv6 address fe80:0000:0000:0000:0302:0304:0506:0708
# hello world.c中输出
```

```
Hello, world
# 使用Ctrl+c 退出
```

控制台中除了输出“Hello, world”之外，还给出了很多提示信息，这些提示信息由native平台下的contiki-main.c中输出，而hello-world.c中仅输出“Hello, world”。下面我们再来分析hello-world.c文件，hello-world.c的具体实现如下：

代码清单10-1 native hello-world.c

```
#include "contiki.h"
#include <stdio.h>
PROCESS(hello_world_process, "Hello world process");
AUTOSTART_PROCESSES(&hello_world_process);
PROCESS_THREAD(hello_world_process, ev, data)
{
    PROCESS_BEGIN();
    printf("Hello, world\n");
    PROCESS_END();
}
```

虽然hello-world.c示例非常简单，但已经包括了protothread机制涉及的常用宏定义——PROCESS(…)、PROCESS_THREAD(…)、PROCESS_BEGIN()、PROCESS_END()和AUTOSTART_PROCESSES(…)。

- PROCESS(hello_world_process, "Hello world process")：PROCESS宏用于任务声明，此处声明一个hello_world_process任务。
- AUTOSTART_PROCESSES(&hello_world_process)：AUTOSTART_PROCESSES宏用于设置“自启动”任务，hello_world_process任务加入到自启动任务列表中，Contiki操作系统完成初始化工作之后便会启动该任务。
- PROCESS_THREAD(hello_world_process, ev, data){}：在任务主体中，任务主体总是以PROCESS_BEGIN()开始并以PROCESS_END()结束，此处hello_world_process任务中并没有while(1)循环结构，所以任务运行一次。

10.3.3 安装交叉工具链

一个入门级别的“Hello World”示例虽然并没有太多的使用价值，但是该示例却展现了使用protothread的基本方法，现在我们可以把native的成功经验迁移到CC2538dk/SensorTag平台中。SensorTag平台和native平台不同，SensorTag平台的开发依赖于gcc-arm-embedded交叉工具链。gcc-arm-embedded交叉工具链的更多信息请参考：https://launchpad.net/gcc-arm-embedded。下面介绍gcc-arm-embedded工具链安装的两种方法：Ubuntu PPA方法和软件包直接安装法。

1. Ubuntu PPA方法

如果使用Ubuntu发行版，可通过Ubuntu PPA服务安装较新版本的gcc-arm-embedded

交叉工具链，在控制台中依次输入以下指令：

```
# Instant Contiki中已经预装了gcc-arm-embedded工具链
# 可使用apt-get remove指令卸载该旧工具链
sudo apt-get remove gcc-arm-none-eabi
# 增加gcc-arm-embedded软件源仓库
sudo add-apt-repository ppa:team-gcc-arm-embedded/ppa
# 更新软件源
sudo apt-get update
# 安装gcc-arm-embedded
sudo apt-get install gcc-arm-embedded
```

2. 软件包直接安装法

其他 Linux 发行版也可以通过下载软件包的方式安装 gcc-arm-embedded 交叉工具链。下面以 gcc-arm-none-eabi 5.4 版本为例说明安装 gcc-arm-embedded 的具体方法。

```
# 获取gcc-arm-embedded软件安装包
wget https://launchpadlibrarian.net/287101520/gcc-arm-none-eabi-5_4-2016q3-20160926-
    linux.tar.bz2
# 复制安装包到/usr/local目录
sudo cp gcc-arm-none-eabi-5_4-2016q3-20160926-linux.tar.bz2 /usr/local
# 在/usr/local目录下解压软件安装包
cd /usr/local
sudo tar -jxvf gcc-arm-none-eabi-5_4-2016q3-20160926-linux.tar.bz2
```

把 gcc-arm-none-eabi-5_4 解压至 /usr/local 之后，并不能立即使用 gcc-arm-embedded 工具链，还需要把交叉工具链的具体路径加入到用户环境变量中。

```
# 使用nano打开bashrc
sudo nano ~/.bashrc
# 在bashrc文件的最后一行增加
PATH=$PATH:/usr/local/gcc-arm-none-eabi-5_4-2016q3/bin
# 退出nano
# 立即更新环境变量
source ~/.bashrc
```

3. 安装验证

完成交叉工具链安装之后，在控制台中输入“ arm-none-eabi-gcc -v”便可查看该工具是否安装成功，若交叉工具链安装成功可在控制台中看到如图 10-8 所示内容。

10.3.4　SensorTag 入门示例

完成了交叉工具链的安装之后，我们立马着手 SensorTag 平台上的“ Hello World”示例。SensorTag 的“Hello World”示例比 native 的“Hello World”示例更复杂一些。

1. 修改 makefile.user.include

与设备相关的代码均位于本书代码仓库 the_beginning_of_coap/microsystem_device 目录中，该目录下有一个名为 makefile.user.include 的文件，该文件用于指定 Contiki 源代码的具

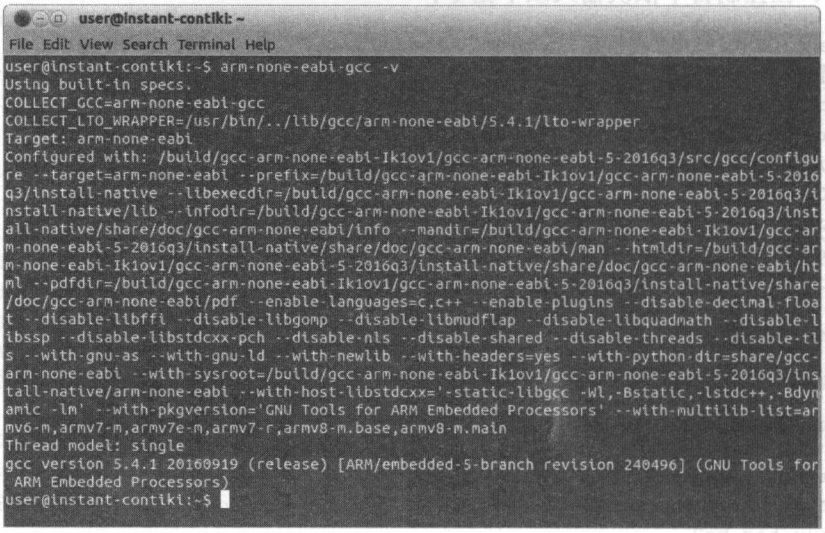


图 10-8　验证 arm-none-eabi-gcc 工具链

体位置，在前面的小节中 Contiki 源代码被复制至 user 用户根目录中，那么 Contiki 源代码的绝对位置为 /home/user/contiki。请根据实际情况修改 Contiki 源代码的具体位置，否则将导致 SensorTag 入门示例和其他所有示例编译失败。makefile.user.include 的具体内容如下：

```
APPDIRS+=$(USER_FOLDER)/apps
TARGETDIRS+=$(USER_FOLDER)/platform

#指定contiki源代码目录位置
# 请务必根据实际情况修改
CONTIKI=/home/user/contiki
include $(CONTIKI)/Makefile.include
```

2. hello-world.c

SensorTag 入门示例位于 /examples/sensortag/hello-world 目录中，该文件夹中包括：hello-world.c、project-conf.h、Makefile.target 和 Makefile 四个文件，其中 hello-world.c 文件的具体内容如下：

代码清单10-2　SensorTag hello-world.c

```
#include "contiki.h"
#include "dev/leds.h"
#include "net/ipv6/uip-ds6.h"
#include <stdio.h>

PROCESS(hello_world_process, "hello world process");
PROCESS(simple_process, "simple process");
AUTOSTART_PROCESSES(&hello_world_process);
```

```
PROCESS_THREAD(hello_world_process, ev, data)
{
    static struct etimer et_red;
    PROCESS_BEGIN();

    etimer_set(&et_red, CLOCK_SECOND / 8);
    printf("hello world!\n");
    while(1) {
        PROCESS_YIELD();

        if(ev == PROCESS_EVENT_TIMER && etimer_expired(&et_red)) {
            if(uip_ds6_get_global(ADDR_PREFERRED) != NULL) {
                leds_off(LEDS_RED);
                printf("device has joined the net\n");
                process_start(&simple_process, NULL);
            } else {
                leds_toggle(LEDS_RED);
                etimer_set(&et_red, CLOCK_SECOND / 8);
            }
        }
    }

    PROCESS_END();
}
PROCESS_THREAD(simple_process, ev, data)
{
    static struct etimer et_green;
    PROCESS_BEGIN();

    printf("simple process\n");
    etimer_set(&et_green, CLOCK_SECOND / 4);

    while(1) {
        PROCESS_YIELD();
        if(ev == PROCESS_EVENT_TIMER && etimer_expired(&et_green)) {
            leds_toggle(LEDS_GREEN);
            etimer_set(&et_green, CLOCK_SECOND / 4);
        }
    }
    PROCESS_END();
}
```

hello-world.c 中包含两个任务：hello_world_process 和 simple_process。hello_world_process 任务周期性查看设备是否已经获得 IPv6 网络前缀，如果暂未分配到网络前缀，则令红色 LED 不停闪烁；如果已经分配到网络前缀，那么关闭红色 LED 并启动 simple_process 任务。

- PROCESS(hello_world_process, "hello world process")：声明 hello_world_process 任

务，并把该任务设置为自启动。

- PROCESS(simple_process, "simple process")：声明 simple_process 任务。
- static struct etimer et_red; etimer_set(&et_red, CLOCK_SECOND / 8)：在 hello_world_process 任务中增加一个 etimer 定时器，该定时器为 Contiki 中的一个默认组件，定时器的溢出时间为 CLOCK_SECOND / 8 也就是 125ms。
- PROCESS_YIELD()：任务挂起等待系统时间。
- if(ev == PROCESS_EVENT_TIMER && etimer_expired(&et_red))：任务获得一个定时器事件，且定时器 etimer 为在该任务中创建的 et_red 定时器。Contiki 中通过 etimer 实现任务的周期性运行。
- uip_ds6_get_global(ADDR_PREFERRED) != NULL：获取全局 IPv6 网络前缀，该前缀由边界路由分配。如果该设备未分配到 IPv6 网络前缀，再次启动 et_red 定时器，超时依然为 CLOCK_SECOND / 8。
- process_start(&simple_process, NULL)：如果分配到 IPv6 网络前缀，则启动 simple 任务。

simple_process 任务是典型的周期性处理任务，但是该任务并不会开机自动，该任务是否启动取决于设备是否已经获取 IPv6 网络前缀。

3. 生成 hello-world.hex

下面我们尝试生成 HEX 格式固件，在控制台中输入：

```
# 进入hello-world示例目录
cd examples/sensortag/hello-world
# 生成固件
make BOARD=sensortag/cc2650
```

在 hello-world 目录下将生成 hello-world.hex 文件，该文件便是 hello-world 示例的最终固件。若在 Instant Contiki 3.0 中生成固件，可以在 Instant Contiki 3.0 中复制 hello-world.hex，切换至 Windows 操作系统后直接粘贴到合适的目录即可；若使用其他 Linux 发行版需要在虚拟机中安装 VMware Tools 工具，该工具可实现 Linux 虚拟机和 Windows 主机之间的复制与粘贴操作。Linux 虚拟机和 Windows 主机之间传输的文件的方法还包括 FTP 和共享文件夹等方法。

4. 下载 hello-world.hex

SensorTag/CC2650 的固件下载操作需要借助 XDS110 下载工具（见图 10-9）和 Flash Programmer2 下载软件。相关软件与工具的详细介绍和使用方法请参考德州仪器官方网站。

- XDS110: http://processors.wiki.ti.com/index.php/Debug_DevPack_User_Guide
- Flash Programmer2: http://www.ti.com.cn/tool/cn/flash-programmer

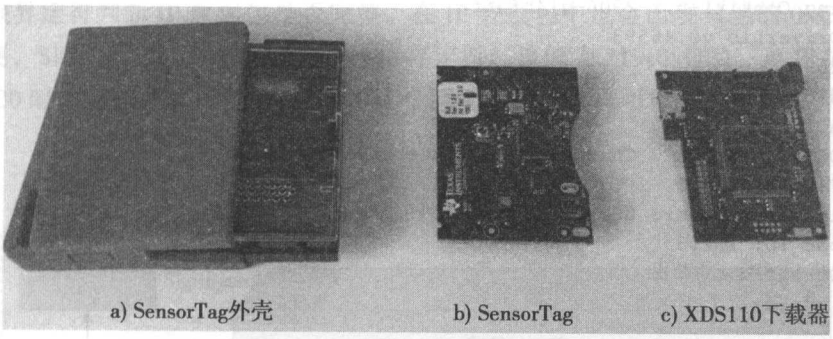


图 10-9　SensorTag 和 XDS110 下载器

为了顺利下载固件首先需要拆除 SensorTag 的保护外壳，把 SensorTag 底板与 XDS110 下载器相连，然后通过 USB 连接线与 Windows 主机相连。打开 Flash Programmer2 下载软件把 hello-world.hex 文件下载至 SensorTag 中，具体操作过程如图 10-10 所示。HEX 文件下载完成之后 SensorTag 将自动重启。

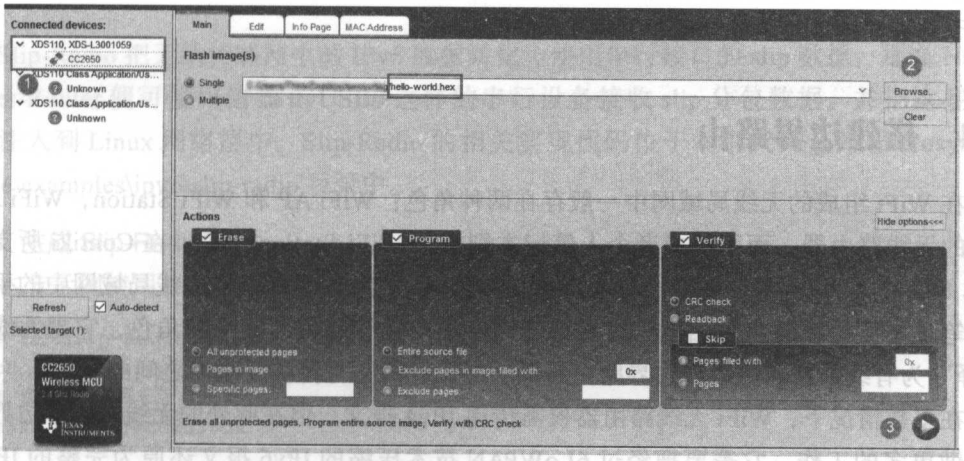


图 10-10　下载 hello-world.hex

5. 运行 hello-world.hex

固件下载完成之后，SensorTag 将进入正常运行模式，XDS110 下载工具将在 Windows 主机中虚拟一个串口设备，利用该串口设备可以观察 SensorTag 的运行情况，通过串口调试工具可以观察到以下类似输出信息：

```
# contiki-main.c中输出
Starting Contiki-3.x-2906-g14bfaff
With DriverLib v0.46593
TI CC2650 SensorTag
```

```
Starting Contiki-3.x-2906-g14bfaff
With DriverLib v0.46593
TI CC2650 SensorTag
Net: sicslowpan
    MAC: CSMA
    RDC: ContikiMAC, Channel Check Interval: 16 ticks
    RF: Channel 25
    Link layer addr: 00:12:4b:00:07:c9:4b:03
    Node ID: 19203
# hello-world.c中输出
hello world!
```

SensorTag 的串口输出信息大致可分为两部分：一部分为 contiki-main.c 输出内容，另一部分为 hello-world.c 输出内容。contiki-main.c 中输出了当前 Contiki 的版本编号，CC2650 相关 driverlib 的版本编号、Platform 名称、MAC 层和 RDC 层选用组件情况、射频部分工作信号、CC2650 链路层 MAC 地址等基本信息。hello-world.c 中的输出信息则相对简单，仅包括“hello world！”。

此时 SensorTag 的红色 LED 将会不停地闪烁，由于并没有边界路由设备，所以 SensorTag 始终无法加入网络，也无法获得全局网络前缀。SensorTag 与外部网络建立联系必须要依赖于边界路由。

10.4 搭建边界路由

在 WiFi 组成的无线局域网中一般存在两种角色：WiFi AP 和 WiFi Station，WiFi AP 即俗称的无线路由器，而手机或者个人笔记本则扮演 WiFi Station 角色。在 Contiki 所支持的 IEEE 802.15.4 无线局域网中，SensorTag 扮演终端设备，这和 WiFi 无线局域网中的手机和个人笔记本所扮演的角色相似。边界路由则扮演与无线路由器相似的角色，它把无线数据“翻译”为有线数据，可以说边界路由是低功耗无线网络与传统互联网之间的桥梁。

在多数情况下，WiFi 无线路由器仅需传递 IPv4 报文，但在此处的无线网络中边界路由需要做更多的工作，它需要把经过 6LoWPAN 技术压缩的 IPv6 报文还原为完整的 IPv6 报文。由于具体网络的限制，边界路由还需要把 IPv6 报文转化为合适的 IPv4 报文。

Contiki 中有多种创建边界路由的方法，本小节将介绍 Slip-Radio + Native-Border-Router 方法。本小节中 CC2538 用于实现 Slip-Radio，而树莓派用于实现 Native-Border-Router，本节的主要工作如图 10-11 所示。

10.4.1 创建 Slip-Radio

Slip 是串行线路接口协议的简称，它是一种点对点网络解决方案，Slip 协议是一种非常古老的串行线路接口协议，该协议提供了一种非常简单的封装 IP 数据包的方法。Slip 协议定义一个称为 slip end 的界定符，该值为 192(0xC0)，该界定符被追加到 IP 数据包的末

尾，利用该界定符判断 IP 数据包是否完整。在 IP 数据包中也会出现其他的 0xC0，为了解决这种冲突，Slip 协议又定义了一个 slip esc 界定符，该值为 219(0xDB)。当 IP 发送数据包中包含 0xC0 时，它被自动替换为 0xDB 0xDC 两个字符，这样可以避免与结尾界定符 0xC0 产生冲突。

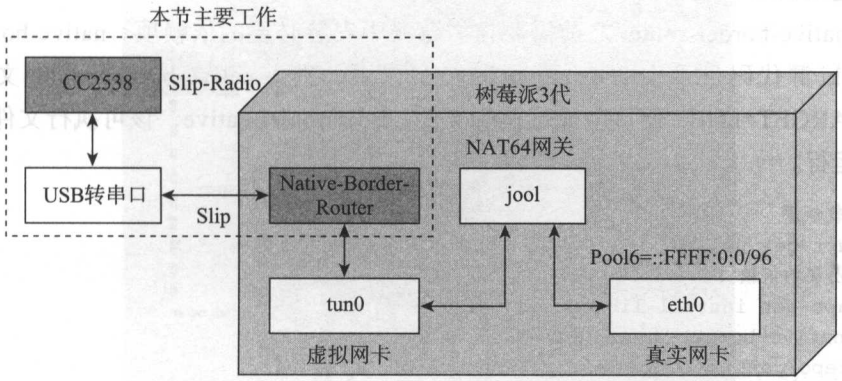


图 10-11　搭建边界路由主要工作

Slip-Radio 把无线传感网中的 IPv6 数据转化为使用串行接口的 slip 数据，那么 Native-Border-Router 便可使用诸如 ttyUSB0 这样的串行设备接收 slip 分包数据，并把这些分包数据注入到 Linux 网络层中。Slip-Radio 的相关实现代码位于本书代码仓库 microsystem_device\examples\ipv6\slip-radio 目录中。

1. 生成 Slip-Radio 固件

本节示例使用 CC2538DK 平台生成 Slip-Radio 固件，具体操作如下：

```
# 进入Slip-Radio目录
cd examples/ipv6/slip-radio
# 在CC2538平台下生成固件
make TARGET=cc2538dk
```

2. 下载 Slip-Radio 固件

CC2538 的固件烧写方式和 SensorTag/CC2650 相似，借助 Flash Programmer2 和 XDS100/XDS110 下载工具便可完成操作。

10.4.2　创建 Native-Border-Router

完成 Slip-Radio 的工作之后，我们接着在树莓派中创建 Native-Border-Router。

1. 树莓派中复制 Contiki 源代码

在树莓派控制台获取 Contiki 源代码，可在树莓派控制台中输入以下内容：

```
# 进入repo目录
```

```
cd ~/repo
# 获取Contiki源代码
git clone https://github.com/contiki-os/contiki.git
# 获取子模块并更新子模块
git submodule update -init
```

2. 创建 Native-Border-Router

编译 native-border-router 之前需要在树莓派中安装必要的依赖项。native-border-router 位于 Contiki 源代码目录中的 examples/ipv6 目录下，进入 native-border-router 文件夹，通过 make TARGET=native 便可生成可执行文件 border-router.native，该可执行文件可在树莓派中直接运行。

```
# 更新软件源
sudo apt-get update
# 安装必要的依赖项
sudo apt-get install libncurses5-dev
# 进入native-border-router目录
cd ~/repo/contiki/examples/ipv6/native-border-router
# 生成可执行文件
make TARGET=native
```

3. border-router.native 使用方法

在树莓派控制台中输入“./border-router.native -h”便可查看 border-router.native 的使用方法，常用的参数如下：

- -B：指定串行通信波特率，默认为 115200。
- -s：指定串行通信设备，默认为 /dev/ttyUSB0。
- -a：用于指定主机 IP 地址，该参数只有在 cooja 仿真才被使用。
- -p：用于指定服务端口号，该参数只有在 cooja 仿真才被使用。
- ipaddress：用于指定全局网络前缀，一般为 fd00::1/64。

```
# 查看border-router.native使用方法
./border-router.native -h
usage:  ./border-router.native [options] ipaddress
example: border-router.native -L -v2 -s ttyUSB1 fd00::1/64
Options are:
    -B baudrate     9600,19200,38400,57600,115200,921600 (default 115200)
    -H              Hardware CTS/RTS flow control (default disabled)
    -L              Log output format (adds time stamps)
    -s siodev       Serial device (default /dev/ttyUSB0)
    -a host         Connect via TCP to server at <host>
    -p port         Connect via TCP to server at <host>:<port>
    -t tundev       Name of interface (default tun0)
```

把 Slip-Radio 通过 USB 转串口设备与树莓派相连，连接方法如图 10-12 所示。一旦 Slip-Radio 与树莓派正确连接，树莓派中将会出现一个名为 ttyUSB0 的设备。

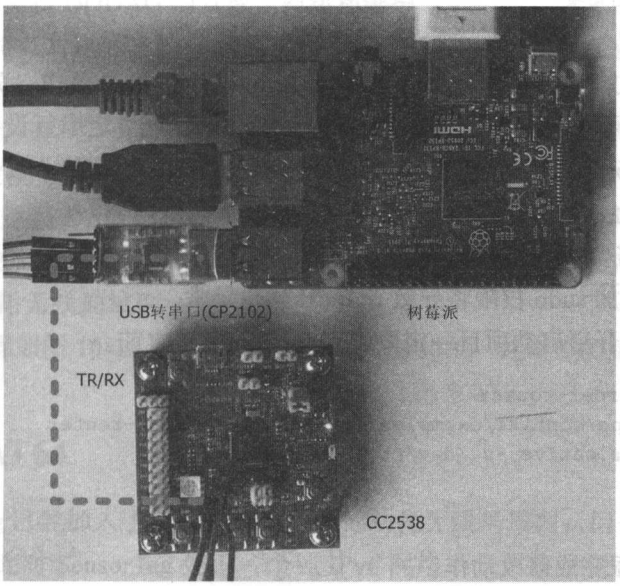


图 10-12　Slip-Radio 与树莓派连接

在树莓派控制台中输入以下命令：

```
sudo ./border-router.native -s /dev/ttyUSB0 fd00::1/64
```

由于需要操作 /dev 目录下的 ttyUSB0 设备，所以此处需要使用 sudo 权限，通过 -s 参数指定使用 ttyUSB0 设备，在此处的 IPv6 网络的全局网络前缀为 fd00::1/64，64 表示网络 ID 为 IPv6 地址的前 64 位。若 border-router.native 正常运行，那么在树莓派控制台中可获得以下输出内容：

```
# 控制台输出
RPL-Border router started
********SLIP started on "/dev/ttyUSB0"
opened tun device "/dev/tun0"
ifconfig tun0 inet `hostname` up
ifconfig tun0 add fd00::1/64
ifconfig tun0
# 省略部分内容
Setting prefix fd00::1
created a new RPL dag
Server IPv6 addresses:
     0x49ba4: =>fd00::212:4b00:5af:8404
     0x49bc4: =>fe80::212:4b00:5af:8404
```

4. 加入开机启动项

为了更方便地使用边界路由，可以把 border-router.native 加入到树莓派开机启动项中，通过编写 /etc/rc.local 文件便可增加用户开机启动项。

```
# 修改树莓派开机启动项
sudo nano /etc/rc.local
```

在/etc/rc.local文件中增加Native-Border-Router启动相关操作。先通过cd指令进入border-router.native可执行文件所在目录，然后通过-s参数指定串行设备名称，一般为“/dev/ttyUSB0”，IPv6全局网络前缀为“fd00::1/64”，指令最后的“&”符号表示border-router.native处于后台运行状态。此处还需要注意以下几点内容：

- 此处虽然需要操作/dev/ttyUSB0设备，但在rc.local文件中并不需要使用sudo权限，盲目增加sudo权限将导致rc.local运行异常。
- 用户增加的开启启动项目一定要出现在“exit 0”之前。

```
# 执行native-border-router
cd /home/pi/repo/contiki/examples/ipv6/native-border-router
./border-router.native -s /dev/ttyUSB0 fd00::1/64 &

exit 0
```

重新启动树莓派完成修改动作。

5. 再次运行SensorTag

重启树莓派之后，SensorTag重新上电并再次运行hello-world，红色LED闪烁一定时间之后便会熄灭。若红色LED停止闪烁说明SensorTag已经加入到由树莓派内Native-Border-Router所建立的低功耗无线局域网中。与SensorTag相连的串口控制台将输出以下内容：

```
# contiki-main.c
TI CC2650 SensorTag
    Net: sicslowpan
    MAC: CSMA
    RDC: ContikiMAC, Channel Check Interval: 16 ticks
    RF: Channel 25
    Link layer addr: 00:12:4b:00:07:c9:4b:03
    Node ID: 19203
# hello-world.c
hello world!
# 成功加入低功耗无线局域网
device has joined the net
simple process
```

低功耗无线局域网的构建依赖于RPL组件，而RPL组件是Contiki的重要组件之一。在构建hello-world固件时已经在其中加入了RPL组件，Contiki操作系统帮助用户完成了终端设备加入低功耗局域网的大多数工作，而用户仅需专注于具体应用便可。RPL机制包括多个基本概念：

- DAG：有向无环图。
- DODAG：面向节点的有向无环图。
- DIO：DAG信息对象，RPL节点发送，用于传递DODAG及其特性信息，该信息

主要被用来进行 DODAG 的发现、构建和维护。

- DAO：支持点对点和点对多的方式沿 DODAG 的上行方向传递目标信息，用于填充前驱节点路由表。

在 RPL 机制中，边界路由定期广播 DIO 请求，终端设备收到 DIO 请求之后，根据 DIO 请求中的内容返回 DAO 响应，边界路由收到 DAO 响应之后便确认某个终端节点已经加入了该低功耗无线局域网。终端节点加入边界路由的过程如图 10-13 所示。

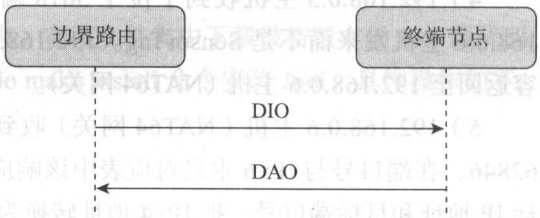


图 10-13　终端节点加入网络过程简述

10.5　增加 NAT64

虽然 SensorTag 已经加入了由树莓派建立的低功耗无线局域网，但是 SensorTag 的联网能力依然不完整。此时 SensorTag 处于一个纯 IPv6 网络中，也就是说 SensorTag 自身只有 IPv6 地址，也只能识别 IPv6 地址，而不能识别任何一个 IPv4 地址。SensorTag 的这种不识别 IPv4 地址的特性看起来非常奇怪，但是这种纯 IPv6 网络确实存在。只能说 IPv6 比 IPv4 简单，而 SensorTag 只能识别更简单的协议。如果与树莓派相连的路由器已经具备 IPv6 功能，并且电信级别运营商也提供 IPv6 功能的话，那么 SensorTag 便可很轻松地访问任意一台具备 IPv6 地址的主机。但是往往事与愿违，IPv4 主机的数量现阶段仍远大于 IPv6 主机。那么 SensorTag 也就只能借助于 NAT64 这样的转换技术才可以间接地访问 IPv4 主机。

10.5.1　NAT64 简介

NAT64 是众多 IPv6 过渡技术的一种，而 Jool 是 Linux 平台下 NAT64 的具体实现，此处通过一个具体的例子说明 NAT64 的基本原理。此时 SensorTag 的 IPv6 地址为 fd00::212:4b00:42a:eaf0，Jool（见 10.5.2 节）部署于树莓派中，而树莓派的 IPv4 地址为 192.168.0.6，SensorTag 需要访问一台 IPv4 主机，IPv4 主机的地址为 192.168.0.3，此时 IPv4 主机在 5678 端口开启一个 UDP ECHO 服务。SensorTag 访问 192.168.0.3 主机的具体过程如图 10-14 所示。

1）SensorTag 访问 IPv4 主机，该主机的 IPv4 地址为 192.168.0.3，由于 SensorTag 无法识别 IPv4 地址，所以采用一种 IPv6-IPv4 兼容地址，IPv4 地址占 IPv6 地址的最后 4 字节，在这 4 字节之前增加一个固定 2 字节前缀 0xffff，其他部分全部使用 0 填充，那么 192.168.0.3 的 IPv6 兼容地址为 ::ffff:192.168.0.3，或可表示为 ::ffff:c0a8:3。

2）SensorTag 构造一个请求，在该请求中 IPv6 源地址为 fd00::212:4b00:42a:eaf0，源端口号为 8765；IPv6 目标地址为 ::ffff:192.168.0.3，目标端口号为 5678。

3）边界路由收到该请求后发现该请求的目标地址为一个 IPv6-IPv4 兼容地址，所以把 ffff 前缀去除，恢复为 192.168.0.6；并把该源地址和源端口号修改为 192.168.0.6 和

62846，此时 Jool 会保存一张端口号和 IPv6 主机的对应表，此时 62848 代表 fd00::212:4b00:42a:eaf0 主机中 8765 端口的具体应用。

4）192.168.0.3 主机收到了位于 5678 端口的请求，但是这台主机认为该请求由 192.168.0.6 主机发来而不是 SensorTag。192.168.0.3 主机根据请求内容返回响应，并把响应内容返回至 192.168.0.6 主机（NAT64 网关）。

5）192.168.0.6 主机（NAT64 网关）收到了响应内容，该主机发现响应目标端口号为 62846，在端口号与 IPv6 主机对应表中该响应需转发至 SensorTag，所以该主机再次修改目标 IP 地址和目标端口号，把 IPv4 地址转换为 IPv6-IPv4 兼容地址。

NAT64 机制其实与 IPv4 中常用的 NAT 机制非常相似，通过 NAT64 机制可帮助纯 IPv6 设备访问 IPv4 应用，如图 10-14 所示。

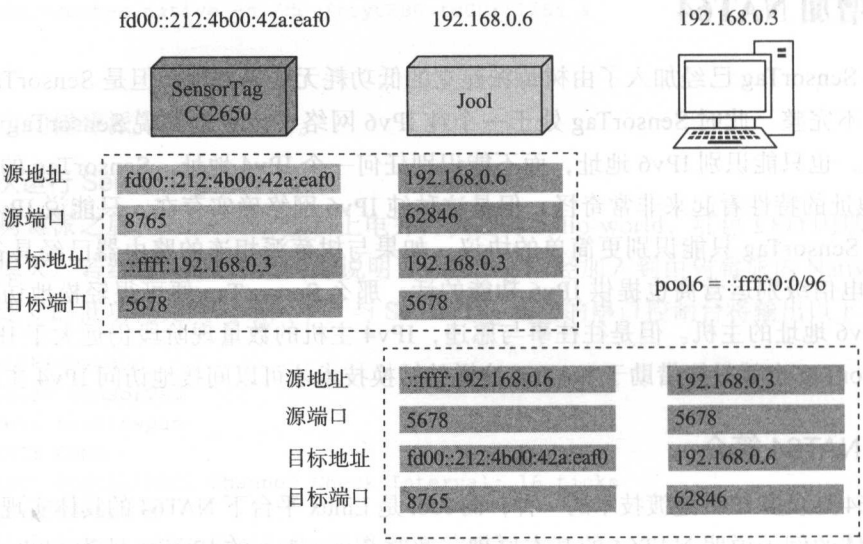


图 10-14　NAT64 机制简介

10.5.2　安装 Jool

了解 NAT64 的原理之后，我们可在树莓派上安装 NAT64 的具体实现——Jool。Jool 是 Linux 平台下 NAT64 的具体实现，Jool 的更多介绍请参考以下网址：

http://www.jool.mx/en/index.html

1. 安装树莓派内核头文件

下面介绍如何在树莓派中安装 Jool，Jool 在 Linux 内核中实现 NAT64，若在树莓派中编译 Jool 需要先安装树莓派内核头文件，在树莓派控制台中输入以下内容：

```
# 更新软件源
sudo apt-get update
```

```
# 安装树莓派内核头文件
sudo apt-get install raspberrypi-kernel-headers
```

2. 编译并安装 Jool

在树莓派中新建一个名为 software 的文件夹，在该文件夹中下载最新版本的 Jool 源代码，然后使用 make 命令编译源代码，最后通过 sudo make install 命令安装 Jool，具体操作如下：

```
# 新建software文件夹
mkdir -p software
cd software
# 下载3.5.1版本Jool源代码，并解压至software目录
wget https://www.jool.mx/download/Jool-3.5.1.zip
unzip Jool-3.5.1.zip
# 进入mod子目录
cd Jool-3.5.1/mod
# 编译Jool源代码
make
# 安装Jool
sudo make install
```

3. 加入开机启动项

为了更方便地启用 NAT64，可在开机启动项目中加载 Jool，与 Native-Border-Router 功能相似，可在 /etc/rc.local 文件中增加载入 Jool 所需的指令。

```
sudo nano /etc/rc.local
```

在 /etc/rc.local 文件中增加 modprobe jool pool6=::ffff:0:0/96，此时 Jool 将把以 ::ffff:0:0/96 开头的 IPv6 地址转化为 IPv4 地址。最终 /etc/rc.local 文件将包含以下内容：

```
# 增加NAT64功能
modprobe jool pool6=::ffff:0:0/96
# 执行native-border-router
cd /home/pi/repo/contiki/examples/ipv6/native-border-router
./border-router.native -s /dev/ttyUSB0 fd00::1/64 &

exit 0
```

重新启动树莓派完成 Jool 安装操作，为了验证 SensorTag 是否可以正常访问 IPv4，我们通过一个 UDP NAT64 示例验证 Jool 是否工作正常。

10.5.3　UDP NAT64 示例

下面我们通过一个 UDP 示例验证边界路由和 NAT64 是否工作正常，在这个示例中，Sensor Tag 将试图访问一个位于 IPv4 地址为 192.168.0.3 的主机中的 IPv4 应用，该 IPv4 应用为一个 UDP Echo Server，在第 3 章已经使用 Python 3 编写了一个 UDP Echo Server，此处我们继续使用该 UDP Server。

SensorTag UDP Client 示例：

```
the_beginning_of_coap \microsystem_device\examples\sensortag\nat-udp-client
```

UDP Echo Server 示例：

```
the_beginning_of_coap\review_demo\udp_echo_demo
```

由于 SensorTag 的目标 UDP 端口号为 5678，需要修改 udp-server.py 中的 UDP 服务端口号为 5678。

1. 示例简述

本节的示例与本章的最终示例非常相似，该示例是 CoAP 客户端示例的基础。该示例中 SensorTag 的 IPv6 全局地址为 fd00::212:4b00:42a:eaf0，SensorTag 将试图访问一个 UDP 服务，该服务位于 IPv4 地址为 192.168.0.3 的主机中，该主机中包含一个 UDP Echo 服务，该 Echo 服务部署于 5678 端口。若按下 SensorTag 的左侧按键，那么 SensorTag 将向 Echo 服务器发起 UDP 请求，这个“不同寻常”的 UDP 请求将由无线数据变为有线数据，将由被 6LoWPAN 压缩的 IPv6 数据包变为常见的 IPv4 数据包，经过一系列的变化之后这个“不同寻常”的 UDP 请求变为一个再普通不过的 IPv4 UDP 请求。对于 192.168.0.3 主机而言，SensorTag 与局域网中其他主机完全相同，而对于 SensorTag 而言，它借助 IEEE 802.15.4 和 6LoWPAN 压缩技术“无缝”地接入了传统网络。图 10-15 的加粗虚线说明了数据转移的大致过程。

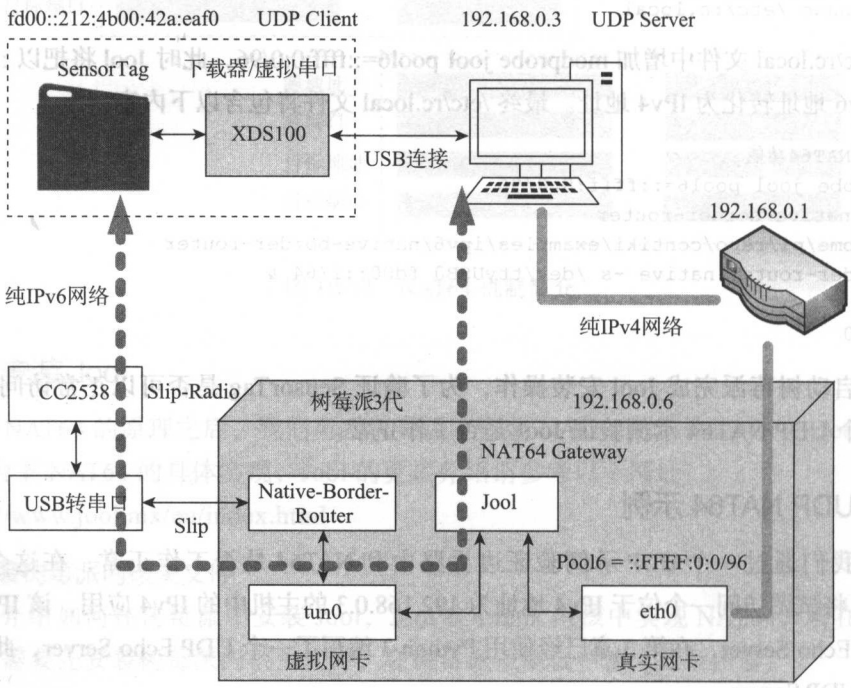


图 10-15　UDP NAT64 示例网路结构图

2. udp-client.c 部分代码

udp-client.c 实现了 UDP Client 功能，具体代码如下：

代码清单10-3　udp-client.c

```
#include "contiki.h"
#include <stdio.h>
#include "net/ip/uip.h"
#include "net/ipv6/uip-ds6.h"
#include "net/ip/uip-udp-packet.h"

#include "dev/leds.h"
#include "dev/button-sensor.h"

#include "ip64-addr.h"

#include <stdio.h>
#include <string.h>

#define UDP_CLIENT_PORT 8765
#define UDP_SERVER_PORT 5678


#define DEBUG DEBUG_PRINT
#include "net/ip/uip-debug.h"

#define MAX_PAYLOAD_LEN      32

static struct uip_udp_conn *client_conn;
static uip_ipaddr_t server_ipaddr;

PROCESS(udp_client_process, "UDP client process");
PROCESS(hello_world_process, "hello world process");
AUTOSTART_PROCESSES(&hello_world_process);

PROCESS_THREAD(hello_world_process, ev, data)
{
    static struct etimer et_red;
    PROCESS_BEGIN();

    etimer_set(&et_red, CLOCK_SECOND / 8);
    printf("hello world!\n");
    while(1) {
        PROCESS_YIELD();

        if(ev == PROCESS_EVENT_TIMER && etimer_expired(&et_red)) {
            if(uip_ds6_get_global(ADDR_PREFERRED) != NULL) {
                leds_off(LEDS_RED);
                printf("device has joined the net\n");
                process_start(&udp_client_process, NULL);
```

```
            } else {
                leds_toggle(LEDS_RED);
                etimer_set(&et_red, CLOCK_SECOND / 8);
            }
        }
    }

    PROCESS_END();
}

static void
tcpip_handler(void)
{
    char *str;
    if(uip_newdata()) {
        str = uip_appdata;
        str[uip_datalen()] = '\0';
        printf("data recv '%s ']\n", str);
    }
}

static void
send_packet(void)
{
    static int seq_id;
    char buf[MAX_PAYLOAD_LEN];

    seq_id++;
    PRINTF("data send to %02X 'Hello %d'\n",
          server_ipaddr.u8[sizeof(server_ipaddr.u8) - 1], seq_id);
    sprintf(buf, "Hello %d \n", seq_id);
    uip_udp_packet_sendto(client_conn, buf, strlen(buf),
                          &server_ipaddr, UIP_HTONS(UDP_SERVER_PORT));
}

PROCESS_THREAD(udp_client_process, ev, data)
{

    PROCESS_BEGIN();

    printf("udp client process started\n");

    uip_ip4addr_t ip4addr;
    uip_ipaddr(&ip4addr, 192, 168, 0, 3);
    ip64_addr_4to6(&ip4addr, &server_ipaddr);

    client_conn = udp_new(NULL, UIP_HTONS(UDP_SERVER_PORT), NULL);
    if(client_conn == NULL) {
        PRINTF("No UDP connection available, exiting the process!\n");
        PROCESS_EXIT();
```

```
    }
    udp_bind(client_conn, UIP_HTONS(UDP_CLIENT_PORT));

    PRINTF("Created a connection with the server ");
    PRINT6ADDR(&client_conn->ripaddr);
    PRINTF(" local/remote port %u/%u\n",
    UIP_HTONS(client_conn->lport), UIP_HTONS(client_conn->rport));
    printf("Press left button to send udp packet");

    while(1) {

        PROCESS_YIELD();
        if(ev == tcpip_event) {
            tcpip_handler();
        }
        if(ev == sensors_event && data == &button_sensor) {
            send_packet();
        }
    }

    PROCESS_END();
}
```

udp-client.c 的具体解释如下。

（1）加入网络后启动 udp_client_process

udp_client_process 并不能上电之后立即运行，该任务需要在 SensorTag 加入网络之后才被启用。加入网络之前红色 LED 不停闪烁，一旦加入网络成功红色 LED 熄灭，并通过串口控制台输出“device has joined the net”。

```
if(uip_ds6_get_global(ADDR_PREFERRED) != NULL) {
    leds_off(LEDS_RED);
    printf("device has joined the net\n");
    process_start(&udp_client_process, NULL);
}
```

（2）构造 IPv4-IPv6 兼容地址

在 Contiki 中可使用 ip64_addr_4to6 构造一个 IPv4-IPv6 兼容地址，此时 server_ipaddr 指向 IPv4 地址为 192.168.0.3 的主机。

```
uip_ip4addr_t ip4addr;
uip_ipaddr(&ip4addr, 192, 168, 0, 3);
ip64_addr_4to6(&ip4addr, &server_ipaddr);
```

（3）构造 UDP 套接字

通过 udp_new 和 udp_bind 构造一个 UDP 客户端套接字。

```
client_conn = udp_new(NULL, UIP_HTONS(UDP_SERVER_PORT), NULL);
udp_bind(client_conn, UIP_HTONS(UDP_CLIENT_PORT));
```

（4）发送 UDP 请求

一旦按下 SensorTag 左侧按键，Contiki 中的传感器驱动便向 udp_client_process 任务发送一个事件，该事件的具体值为 button_sensor。udp_client_process 接收到该事件之后便会执行 UDP 请求。

```
if(ev == sensors_event && data == &button_sensor) {
    send_packet();
}
```

（5）处理 UDP 响应

SensorTag 一旦收到 UDP 响应，Contiki 中的网络驱动便向 udp_client_process 任务传递一个 tcpip_event 事件。

```
if(ev == tcpip_event) {
    tcpip_handler();
}
```

在 tcpip_handler 函数中，SensorTag 从 uip_appdata 中可读取 UDP 响应内容，而 uip_datalen 可指示 UDP 响应内容长度。SensorTag 把响应内容输出到串口控制台中。

```
static void
tcpip_handler(void)
{
    char *str;
    if(uip_newdata()) {
        str = uip_appdata;
        str[uip_datalen()] = '\0';
        printf("data recv '%s ']\n", str);
    }
}
```

3. 测试并验证

下面我们来测试这个不同寻常的 UDP Echo 示例。

（1）生成并下载 udp-client.hex

```
# 生成hex固件
cd ~/resp/the_beginning_of_coap/microsystem_device/examples/sensortag/nat-udp-
    client
make BOARD=sensortag/cc2650
```

通过 make 命令生成 udp-client.hex 文件，并使用 Flash Programmer2 下载软件把固件下载至 SensorTag 中。

（2）在 192.168.0.3 主机中运行 udp-server.py

修改 UDP 侦听端口号为 5678，在 192.168.0.3 主机中运行 udp_server.py（见 3.3.1 节）。

```
# 在192.168.0.3主机中运行udp_server.py
python3 udp_server.py
```

（3）等待 SensorTag 加入网络

SensorTag 重新上电运行之后将加入由树莓派内边界路由组成的低功耗网络，该网络的全局前缀为 fd00::1/64。若 SensorTag 未加入该网络，红色 LED 将持续闪烁，一旦加入网络成功，红色 LED 将停止闪烁，绿色 LED 将处于常亮状态。在串口控制台将获得以下输出：

```
Starting Contiki-3.x-2906-g14bfaff
With DriverLib v0.46593
TI CC2650 SensorTag
    Net: sicslowpan
    MAC: CSMA
    RDC: ContikiMAC, Channel Check Interval: 16 ticks
    RF: Channel 25
    Node ID: 19203
hello world!
device has joined the net
udp client process started
Client IPv6 Address: fd00::212:4b00:7c9:4b03
fe80::212:4b00:7c9:4b03
Server address: ::FFFF:192.168.0.3
Created a connection with the server :: local/remote port 8765/5678
Press left button to send udp packet
```

串口输出的最后提示用户按下 SensorTag 的左侧按键，一旦按下左侧按键 SensorTag 将会启动一次 UDP 请求。

（4）按下 SensorTag 左侧按键

按下左侧按键之后，SensorTag 将会发送一次 UDP 请求，SensorTag 一旦收到服务器响应内容，它将把响应内容输出至串口控制台。UDP 请求中包含一个序号，每按下一次左侧按键，该序号值将递增一次，如第一次按下时 UDP 请求内容为“Hello 1”，第二次按下时 UDP 请求内容为“Hello 2”。串口控制台将获得以下类似内容：

```
data send to 03 'Hello 1'
data recv 'Hello 1'
data send to 03 'Hello 2'
data recv 'Hello 2'
```

10.6　CoAP Client Sensor

经过前面几个小节的准备工作，我们已经熟悉了开发 Contiki 应用的基本方法，另外树莓派内的边界路由和 NAT64 网关也可正常工作。只要在 SensorTag 中增加必要传感器检测功能和 CoAP 客户端功能便可组成一个完整的应用。本小节中 CoAP Client Sensor 示例是一个较为完整的示例，具体实现代码请参考本书代码仓库：

the_beginning_of_coap\microsystem_device\examples\sensortag\coap-client-sensor

CoAP Client Sensor 示例由三个 C 文件组成：

- coap-client-sensor.c 包含一个启动任务，SensorTag 加入网络之后立刻启动相应的传感器检测任务和 CoAP 客户端推送任务。
- sensor-process.c 包含一个传感器检测任务，该任务定期获取温度、湿度和光照传感器检测结果。一旦检测完成，将通过 Contiki 的消息载体传递至 coap-post-process 任务。
- coap-post-process.c 包含一个 CoAP 客户端任务，该任务把从传感器任务获取的最新数据封装为 JSON 格式，并向指定服务器发送 CoAP 请求。

10.6.1 加入网络并启动任务

coap-client-sensor.c 中仅包括 start_process 任务，一旦 SensorTag 加入网络便启动 sensor_process 和 coap_post_process 两个任务。

代码清单10-4 coap-client-sensor.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "contiki.h"
#include "contiki-net.h"

#include "dev/leds.h"
#include "board-peripherals.h"

#include "sensor-process.h"
#include "coap-post-process.h"

#define DEBUG DEBUG_PRINT
#include "net/ip/uip-debug.h"

PROCESS(start_process, "start process");
AUTOSTART_PROCESSES(&start_process);
PROCESS_THREAD(start_process, ev, data)
{
    static struct etimer et_red;
    PROCESS_BEGIN();

    etimer_set(&et_red, CLOCK_SECOND / 8);
    printf("system start!\n");
    while(1) {
        PROCESS_YIELD();
        if(ev == PROCESS_EVENT_TIMER && etimer_expired(&et_red)) {
            if(uip_ds6_get_global(ADDR_PREFERRED) != NULL) {
                leds_off(LEDS_RED);
                printf("device has joined the net\n");
                process_start(&sensor_process, NULL);
                process_start(&coap_post_process, NULL);
```

```
            } else {
                leds_toggle(LEDS_RED);
                etimer_set(&et_red, CLOCK_SECOND / 8);
            }
        }
    }

    PROCESS_END();
}
```

10.6.2　获取传感器数据

sensor_process 主要用于获取传感器检测结果，与 SensorTag 相关的温湿度传感器和光照传感器已经包含在 Contiki 传感器驱动中，只需调用相应的接口便可以获取检测结果。在 Contiki 传感器驱动中，温湿度传感器 HDC1000 和光照传感器 OPT3001 被设计为“异步”传感器。一旦完成传感器检测动作，传感器驱动将向相关任务发送一个完成事件。虽然这样的方式显得有些麻烦，但可以减少 MCU 等待时间，提高执行效率。

代码清单10-5　sensor_process.c

```
#include "sensor-process.h"
#include "coap-post-process.h"

#define DEBUG DEBUG_PRINT
#include "net/ip/uip-debug.h"

static struct etimer et;
process_event_t sensor_data_ready;
sensor_data_t sensor_give;
static void get_hdc_reading()
{
    int value;
    value = hdc_1000_sensor.value(HDC_1000_SENSOR_TYPE_TEMP);
    if(value != CC26XX_SENSOR_READING_ERROR) {
        sensor_give.temp = value / 100;
        printf("HDC: Temp = %d.%02d C\n", value / 100, value % 100);
    }

    value = hdc_1000_sensor.value(HDC_1000_SENSOR_TYPE_HUMIDITY);
    if(value != CC26XX_SENSOR_READING_ERROR) {
        sensor_give.hum = value / 100;
        printf("HDC: Humidity = %d.%02d %%RH\n", value / 100, value % 100);
    }
}
static void get_light_reading()
{
    int value;
    value = opt_3001_sensor.value(0);
```

```
    if(value != CC26XX_SENSOR_READING_ERROR) {
        sensor_give.light = value / 100;
        printf("OPT: Light = %d.%02d lux\n", value / 100, value % 100);
    }
}
PROCESS(sensor_process, "sensor monitoring");
PROCESS_THREAD(sensor_process, ev, data)
{
    PROCESS_BEGIN();
    printf("sensor process start!\n");
    sensor_data_ready = process_alloc_event();

    etimer_set(&et, CLOCK_SECOND * 60);

    while(1)  {
        PROCESS_YIELD();

        if(ev == PROCESS_EVENT_TIMER && data == &et) {
            printf("start to acquire sensor data\n");
            SENSORS_ACTIVATE(opt_3001_sensor);
            etimer_set(&et, CLOCK_SECOND * 60);
        } else if(ev == sensors_event && data == &opt_3001_sensor) {
                get_light_reading();
                SENSORS_ACTIVATE(hdc_1000_sensor);
        } else if(ev == sensors_event && data == &hdc_1000_sensor) {
                get_hdc_reading();
                process_post(&coap_post_process, sensor_data_ready, (void *)&sensor_
                    give);
        }

    }
    PROCESS_END();
}
```

sensor_process.c 的具体说明如下。

（1）间隔 60 s 获取一次传感器结果

sensor_process 通过一个 etimer 定时器间隔 60s 获取一次传感器检测结果。一旦 60s 定时时间到，则先开启光照传感器 SENSORS_ACTIVATE(opt_3001_sensor)。

```
PROCESS_THREAD(sensor_process, ev, data)
{
    etimer_set(&et, CLOCK_SECOND * 60);
    while(1)  {
        PROCESS_YIELD();

        if(ev == PROCESS_EVENT_TIMER && data == &et) {
            printf("start to acquire sensor data\n");
            SENSORS_ACTIVATE(opt_3001_sensor);
            etimer_set(&et, CLOCK_SECOND * 60);
```

```
        }
        // 省略部分代码
    }
    PROCESS_END();
}
```

（2）光照传感器检测完成之后再启动温湿度传感器

若光照传感器检测完成，Contiki 传感器驱动将向任务传递一个 sensors_event 事件，该事件的值为 opt_3001_sensor。通过 get_light_reading 函数获取光照传感器结果之后，再通过 SENSORS_ACTIVATE（hdc_1000_sensor）启动温湿度传感器转换。一旦温湿度传感器检测完成，Contiki 传感器驱动将会向任务传递一个 sensors_event 事件，该事件的值为 hdc_1000_sensor。

```
while(1)  {
    PROCESS_YIELD();
    if(ev == PROCESS_EVENT_TIMER && data == &et) {
        SENSORS_ACTIVATE(opt_3001_sensor);
    } else if(ev == sensors_event && data == &opt_3001_sensor) {
            get_light_reading();
            SENSORS_ACTIVATE(hdc_1000_sensor);
    } else if(ev == sensors_event && data == &hdc_1000_sensor) {
            get_hdc_reading();
            process_post(&coap_post_process, sensor_data_ready, (void *)&sensor_
                give);
    }
}
```

（3）向 coap_post_process 任务传递检测结果

通过 process_post 函数向 coap_post_process 任务传递转换结果。process_post 用于 Contiki 内不同任务间传递事件和事件结果，若温湿度传感器和光照传感器都完成检测，那么通过 sensor_data_ready 事件把传感器检测结果传递至 coap_post_process 任务中。

```
process_post(&coap_post_process, sensor_data_ready, (void *)&sensor_give);
```

10.6.3　传递传感器数据

coap_post_process.c 主要负责把传感器检测结果通过 CoAP 推送至服务器。

代码清单10-6　coap_post_process.c

```
#include "coap-post-process.h"
#include "sensor-process.h"

#define DEBUG DEBUG_PRINT
#include "net/ip/uip-debug.h"

#define LOCAL_PORT      UIP_HTONS(COAP_DEFAULT_PORT + 1)
#define REMOTE_PORT     UIP_HTONS(COAP_DEFAULT_PORT)
```

```
uip_ipaddr_t server_ipaddr;

void
client_chunk_handler(void *response)
{
    const uint8_t *chunk;
    int len = coap_get_payload(response, &chunk);
    printf("coap response payload:\n");
    printf("[%d]%s\n", len, (char *)chunk);
}
PROCESS(coap_post_process, "coap client");
PROCESS_THREAD(coap_post_process, ev, data)
{
    PROCESS_BEGIN();
    printf("coap post process start!\n");

    static coap_packet_t request[1];

    uip_ip4addr_t ip4addr;
    uip_ipaddr(&ip4addr, 192, 168, 0, 3);
    ip64_addr_4to6(&ip4addr, &server_ipaddr);
    printf("Server address: ");
    print6addr(&server_ipaddr);
    printf("\n");

    uint8_t dev_addr[8];
    ieee_addr_cpy_to(dev_addr, 8);

    static char url[32];
    sprintf(url, "devices/%02X%02X", dev_addr[6], dev_addr[7]);
    printf("device url: %s\n", url);

    /* receives all CoAP messages */
    coap_init_engine();

    while(1) {
        PROCESS_YIELD();
        if (ev == sensor_data_ready) {
            printf("take sensor date ready event\n");
            sensor_data_t sensor_take = *(sensor_data_t*)data;
            coap_init_message(request, COAP_TYPE_CON, COAP_POST, 0);
            coap_set_header_uri_path(request, url);
            coap_set_header_content_format(request, APPLICATION_JSON);

            char payload[32];
            memset(payload, 0x00, 32);
            int len = sprintf(payload, "{\"temp\":%d, \"hum\":%d, \"light\":%d}",
                              sensor_take.temp, sensor_take.hum, sensor_take.
                                  light);
```

```
            coap_set_payload(request, (uint8_t *)payload, len);

            printf("coap request payload: %s\n", payload);
            print6addr(&server_ipaddr); printf(" : %u\n", UIP_HTONS(REMOTE_PORT));

            COAP_BLOCKING_REQUEST(&server_ipaddr, REMOTE_PORT, request,
                                  client_chunk_handler);
        }
    }

    PROCESS_END();
}
```

coap_post_process.c 的具体解释如下。

（1）构造 IPv4-IPv6 兼容地址

在 Contiki 操作系统中可使用 ip64_addr_4to6 构造一个 IPv4-IPv6 兼容地址，此时 server_ipaddr 指向 IPv4 地址为 192.168.0.3 的主机。

```
uip_ip4addr_t ip4addr;
uip_ipaddr(&ip4addr, 192, 168, 0, 3);
ip64_addr_4to6(&ip4addr, &server_ipaddr);
```

（2）获取网卡地址并构造 URL

在微型物联网系统服务器部分已经定义，设备的编号由 4 个数字组成，为了尽可能保证每个设备编号不相同，此处使用 SensorTag/CC2650 的 MAC 地址的最后两字节作为设备编号，使用 ieee_addr_cpy_to 函数可获取 CC2650 的 MAC 地址。

```
uint8_t dev_addr[8];
ieee_addr_cpy_to(dev_addr, 8);
static char url[32];
sprintf(url, "devices/%02X%02X", dev_addr[6], dev_addr[7]);
```

（3）构造 CoAP 请求首部

通过 coap_init_message 函数设置 CoAP 首部中报文类型和 CoAP 方法，根据微型物联网系统服务器部分的设计，CoAP 请求类型为 CON，CoAP 请求方法为 POST；此处还通过 coap_set_header_uri_path 函数定义 CoAP 首部中的 URL，通过 coap_set_header_content_format 定义 CoAP 请求媒体类型为 JSON 类型。

```
coap_init_message(request, COAP_TYPE_CON, COAP_POST, 0);
coap_set_header_uri_path(request, url);
coap_set_header_content_format(request, APPLICATION_JSON);
```

（4）构造 CoAP 请求负载

通过 sprintf 函数构造 JSON 格式的负载，并通过 coap_set_payload 设置本次 CoAP 请求的具体负载，最后通过 COAP_BLOCKING_REQUEST 写入到 Contiki 网络层驱动中。

```
char payload[32];
```

```
memset(payload, 0x00, 32);
int len = sprintf(payload, "{\"temp\":%d, \"hum\":%d, \"light\":%d}",
        sensor_take.temp, sensor_take.hum, sensor_take.light);
coap_set_payload(request, (uint8_t *)payload, len);
COAP_BLOCKING_REQUEST(&server_ipaddr, REMOTE_PORT, request,
                      client_chunk_handler);
```

10.7 综合测试

最后我们进行一个较为综合的测试验证终端节点是否正常工作，本章示例或许是本书中最为复杂的示例，该示例涉及 Contiki 系统的基本使用方法、边界路由的创建、NAT64 实现等部分。综合测试可分为三步实现：

1）在 192.168.0.3 主机中运行“node app.js”，启用 CoAP 服务器和 Web 服务器。

2）在树莓派中启动边界路由和 NAT64。

3）在 SensorTag 中下载 coap-client-sensor.hex 固件。

综合测试的具体流程如图 10-16 所示。

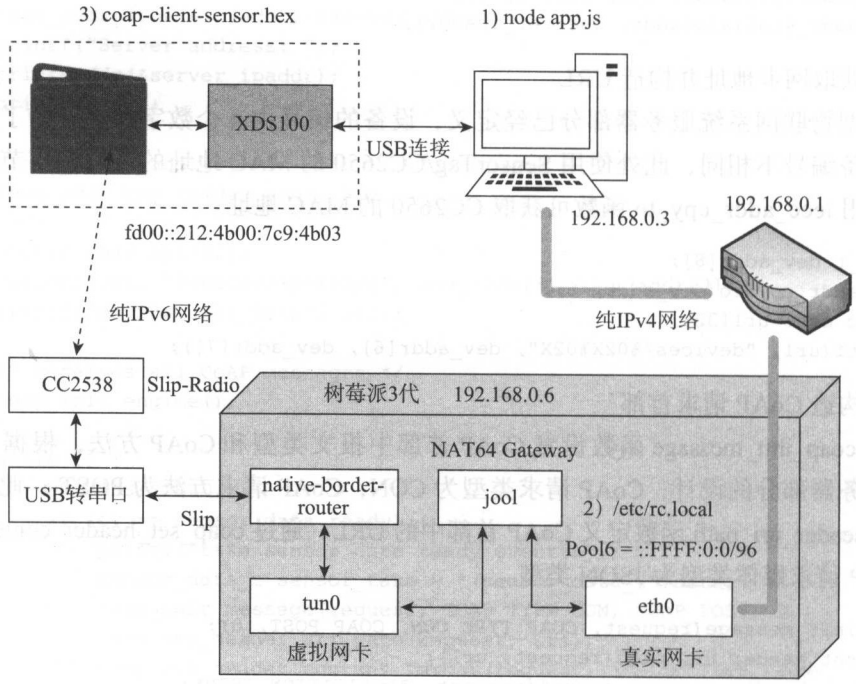


图 10-16 综合测试具体流程

10.7.1 启动 CoAP 服务器

在第 9 章已经完成服务器的开发工作，为了完成此处的综合测试，需要保证 192.168.0.3 主

机中已经正确安装了 MySQL 数据库和 Node.js。该部分的详细实现代码可参考本书代码仓库：

the_beginning_of_coap\microsystem_servernode\app.js

在测试服务器主机 (192.168.0.3) 内启用 CoAP 服务器和 Web 服务器，在控制台中输入以下内容：

```
node app.js
```

10.7.2　启动边界路由和 NAT64

在树莓派中正确启用边界路由和 NAT64 功能。在树莓派 /etc/rc.local 文件中需要包含以下类似内容：

```
# 增加NAT64功能
modprobe jool pool6=::ffff:0:0/96
# 执行native-border-router
cd /home/pi/repo/contiki/examples/ipv6/native-border-router
./border-router.native -s /dev/ttyUSB0 fd00::1/64 &

exit 0
```

10.7.3　生成并下载固件

把 coap-client-sensor.hex 固件下载至 SensorTag 中。

```
# 进入指定目录
cd
~/resp/the_beginning_of_coap/microsystem_device/examples/sensortag/coap-client-
sensor
# 生成固件，并把固件下载至SensorTag中
make BOARD=sensortag/cc2650
```

10.7.4　查看运行结果

当 SensorTag 正常运行之后，我们再通过串口控制台、Node.js 控制台和网页等手段查看系统是否正常运行。

1. SensorTag 串口控制台输出

在 SensorTag 串口控制台中我们可以观察到以下类似输出：

```
TI CC2650 SensorTag
    Net: sicslowpan
    MAC: CSMA
    RDC: ContikiMAC, Channel Check Interval: 16 ticks
    RF: Channel 25
    Link layer addr: 02:12:4b:00:07:c9:4b:03
    Node ID: 19203
system start!
```

```
device has joined the net
sensor process start!
coap post process start!
Server address: ::FFFF:192.168.0.3
device url: devices/4B03
start to acquire sensor data
OPT: Light = 67.19 lux
HDC: Temp = 26.38 C
HDC: Humidity = 52.36 %RH
take sensor date ready event
coap request payload: {"temp":26, "hum":52, "light":67}
::FFFF:192.168.0.3 : 5683
coap response payload:
[17]{"ts":1478355804}
start to acquire sensor data
OPT: Light = 68.48 lux
HDC: Temp = 26.30 C
HDC: Humidity = 52.14 %RH
take sensor date ready event
coap request payload: {"temp":26, "hum":52, "light":68}
::FFFF:192.168.0.3 : 5683
coap response payload:
[17]{"ts":1478355864}
```

此时 SensorTag 完成了以下内容：

1）device has joined the net：SensorTag 成功加入网络，此时红色 LED 停止闪烁。

2）device url: devices/4B03：SensorTag 对应的设备编号为 4B03，此时 SensorTag 应向 coap://192.168.0.3/devices/4B03 推送传感器检测结果。

3）SensorTag 完成了两次传感器数据采集，并把检测结果封装为 JSON 格式。由于间隔的时间较短，所以两次的温湿度和光照检测结果几乎相同，温度检测结果为 26，湿度检测结果为 52，光照检测结果为 68。

4）SensorTag 向 ::FFFF:192.168.0.3 主机发送 CoAP 请求，此时 ::FFFF:192.168.0.3 为一个 IPv4-IPv6 兼容地址，实际指向 IPv4 地址为 192.168.0.3 的主机。

2. Node.js 控制台输出

在 Node.js 控制台中也可以观察到传感器推送数据，此时 CoAP 服务器接收到两次 CoAP 请求，通过与串口控制台的输出结果比较可以发现 CoAP 服务器收到了正确的传感器结果。Node.js 控制台的输出结果如图 10-17 所示。

3. 网页显示

最后使用浏览器查看 4B03 设备 (SensorTag) 的历史结果，在浏览器中输入 http://192.168.0.3:8090，在左侧设备列表中选择“4B03”便可观察到该设备的所有历史数据。4B03 设备的所有历史数据如图 10-18 所示，CoAP 服务器将两次 CoAP 请求中的传感器记录插入至数据库中，其结果与 SensorTag 串口控制台的输出内容相符。

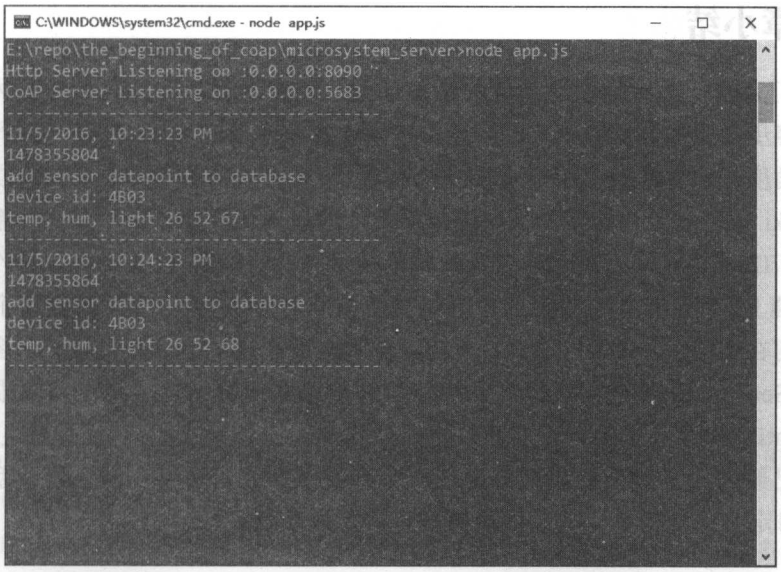


图 10-17　Node. js app.js 控制台输出

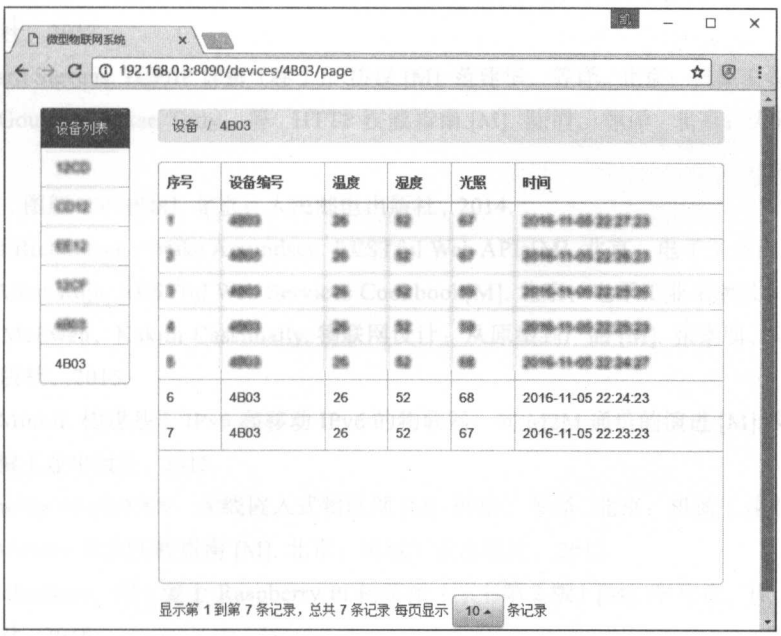


图 10-18　SensorTag 4B03 设备历史结果

通过 SensorTag 串口控制台、Node.js 控制台和网页显示结果，我们可以确认微型物联网系统已经能够正常运行。

10.8 本章小结

本章我们完成了微型物联网系统的设备部分，相比于服务器部分的开发内容，设备部分的开发内容步骤更多。从嵌入式硬件角度来说，本章主要在 SensorTag 中进行开发，从嵌入式软件的角度来说，本章主要通过 Contiki 实现传感器检测和 CoAP 客户端功能。在物联网领域 Contiki 是一个为低功耗受限制设备而设计的嵌入式操作系统，该操作系统包括完整的 6LoWPAN/IPv6 协议栈，本章通过几个简单的示例说明如何使用 Contiki 操作系统。

在一个纯 IPv6 网络中终端设备并不能直接访问 IPv4 应用，为了解决这个矛盾，本章还介绍了如何在树莓派中启用 NAT64 功能。最后本章通过一个完整的 CoAP 客户端示例说明如何使用 SensorTag 向指定 CoAP 服务器推送传感器检测结果。由于篇幅限制，本章不能够展示 Contiki 操作系统的方方面面，但此处的 SensorTag CoAP 已经具有足够的代表性。至此，我们已经完成了微型物联网系统的所有开发工作，只要符合服务器微型物联网系统的 CoAP 接口定义，任何真实或虚拟设备均可接入该系统。

参考文献

[1] Jean-Philippe Vasseur, Adam Dunkels. 基于 IP 的物联网架构、技术与应用 [M]. 田辉，译 . 北京：人民邮电出版社，2011 .

[2] Joseph Davies. 深入解析 IPv6（原书第 3 版）[M]. 北京：人民邮电出版社，2014.

[3] Sergio Scaglia. 嵌入式 Internet TCP/IP 基础、实现及应用 [M]. 北京：北京航空航天大学出版社，2008.

[4] 竹下隆史，村山公宝，等 . 图解 TCP/IP（原书第 5 版）[M]. 乌尼日其其格，译 . 北京：人民邮电出版社，2013.

[5] W Richard Stevens. TCP/IP 详解（卷 1）：协议 [M]. 范建华，等译 . 北京：机械工业出版社，2012.

[6] David Gourley，Brian Totty，等 . HTTP 权威指南 [M]. 陈涓，等译 . 北京：人民邮电出版社，2012.

[7] 上野 宣 . 图解 HTTP[M]. 北京：人民邮电出版社，2014.

[8] Leonard Ricbardson，Mike Amundsen. RESTful Web APIs[M]. 北京：电子工业出版社，2014.

[9] Subbu Allamaraju. RESTful Web Services Cookbook[M]. 北京：电子工业出版社，2013.

[10] Adrian McEwen，Kakim Cassimally. 物联网设计：从原型到产品 [M]. 张崇明，译 . 北京：人民邮电出版社，2015.

[11] Daniel Minoli. 构建基于 IPv6 和移动 IPv6 的物联网：向 M2M 通信的演进 [M]. 郎为民，译 . 北京：机械工业出版社，2015.

[12] Zach Shelby. 6LoWPAN：无线嵌入式物联网 [M]. 韩松，等译 . 北京：机械工业出版社，2015.

[13] 程晨 . Arduino 开发实战指南 [M]. 北京：机械工业出版社，2012.

[14] Matt Richardson，等 . 爱上 Raspberry Pi 树莓派（原书第 2 版）[M]. 李凡希，译 . 北京：人民邮电出版社，2016.

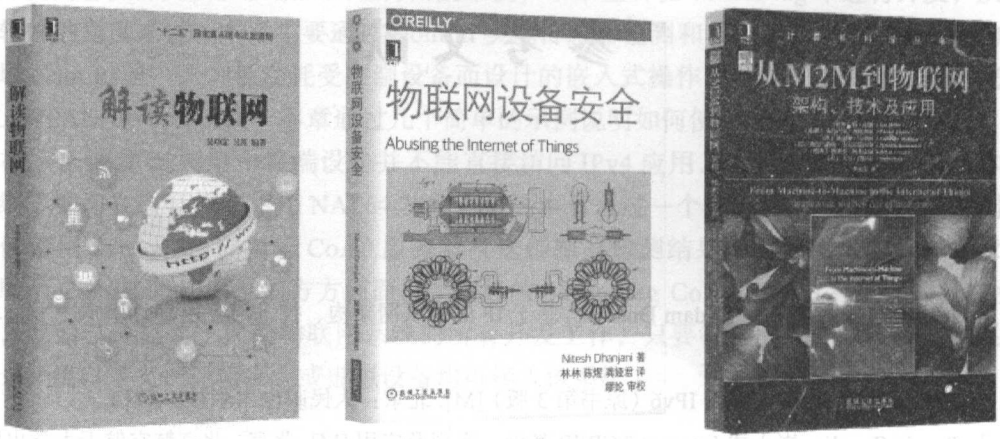
[15] Simon Monk. Raspberry Pi 开发实战 [M]. 黄鑫，译 . 北京：机械工业出版社，2015.

[16] Miguel Grinberg. Flask Web 开发 [M]. 安道，译 . 北京：人民邮电出版社，2015.

[17] Guillermo Rauch. 了不起的 Node.js[M]. Goddy Zhao，译 . 北京：电子工业出版社，2014.

[18] 黄缙华 . MySQL 入门很简单 [M]. 北京：清华大学出版社，2011.

推荐阅读



解读物联网

作者：吴功宜 吴英 ISBN：978-7-111-52150-1 定价：79.00元

本书采用“问/答”形式，针对物联网学习者常见的困惑和问题进行解答。通过全书300多个问题，辅以400余幅插图以及大量的数据、表格，深度解析了物联网的背景知识和疑难问题，帮助学习者理解物联网的方方面面。

物联网设备安全

作者：Nitesh Dhanjani 等 ISBN：978-7-111-55866-8 定价：69.00元

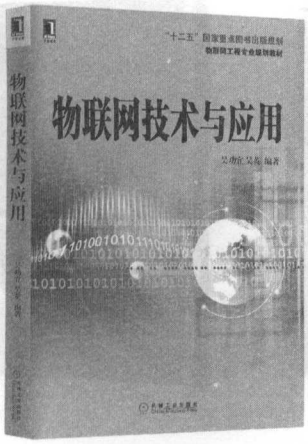
未来，几十亿互联在一起的“东西”蕴含着巨大的安全隐患。本书向读者展示了恶意攻击者是如何利用当前市面上流行的物联网设备（包括无线LED灯泡、电子锁、婴儿监控器、智能电视以及联网汽车等）实施攻击的。

从M2M到物联网：架构、技术及应用

作者：Jan Holler 等 ISBN：978-7-111-54182-0 定价：69.00元

本书由长期从事M2M和物联网领域研发的技术和商务专家撰写，他们致力于从不同视角勾画出一个完整的物联网技术体系架构。书中全面而又详实地论述了M2M和物联网通信与服务的关键技术，以及向物联网演进的过程中所要应对的挑战与需求，同时还介绍了主要的国际标准和一些业界最新研究成果。本书在强调概念的同时，通过范例讲解概念和相关的技术，力求进行深入浅出的阐明和论述。

推荐阅读

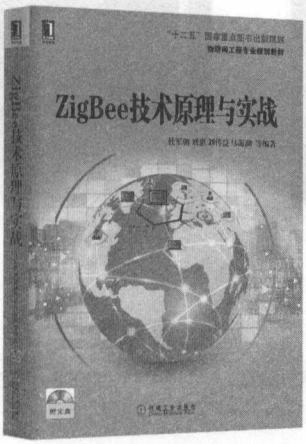


物联网技术与应用

作者：吴功宜 等 ISBN：978-7-111-43157-2 定价：35.00元

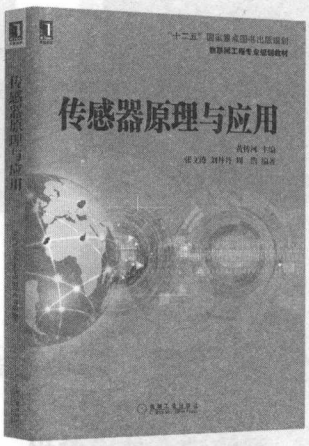
物联网信息安全

作者：桂小林 等 ISBN：978-7-111-47089-2 定价：45.00元



ZigBee技术原理与实战

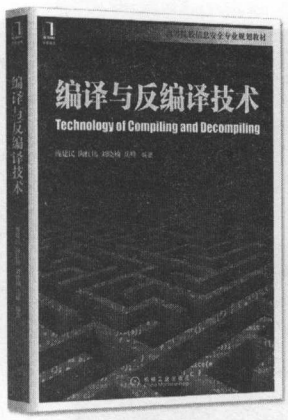
作者：杜军朝 等 ISBN：978-7-111-48096-9 定价：59.00元

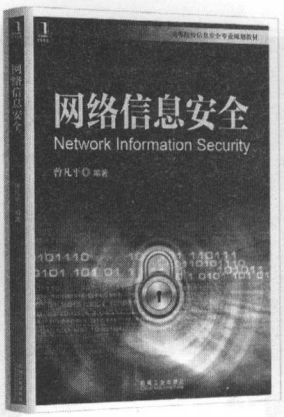


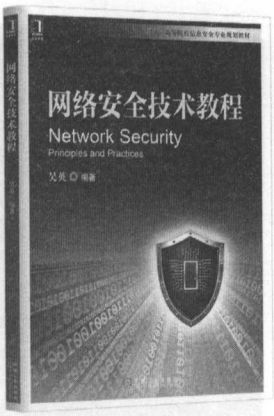
传感器原理与应用

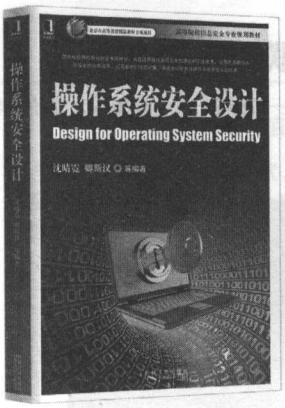
作者：黄传河 ISBN：978-7-111-48026-6 定价：35.00元

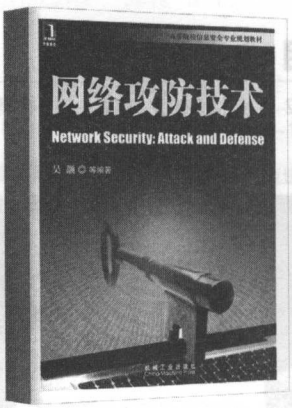
推荐阅读

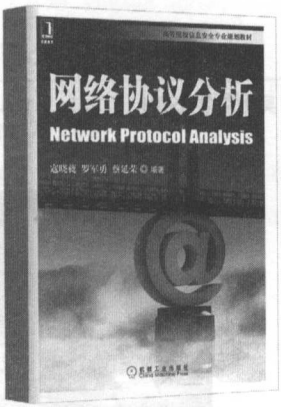












编译与反编译技术

作者：庞建民 等 ISBN：978-7-111-53412-9 定价：59.00元

网络信息安全

作者：曾凡平 ISBN：978-7-111-52008-5 定价：45.00元

网络安全技术教程

作者：吴英 ISBN：978-7-111-51741-2 定价：35.00元

操作系统安全设计

作者：沈晴霓 等 ISBN：978-7-111-43215-9 定价：59.00元

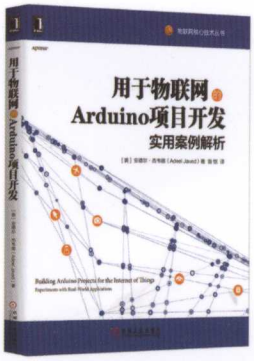
网络攻防技术

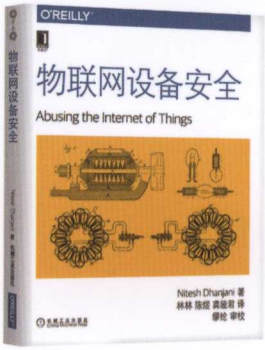
作者：吴灏 等 ISBN：978-7-111-27632-6 定价：29.00元

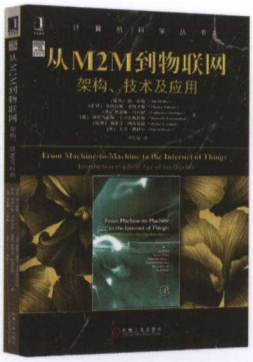
网络协议分析

作者：寇晓蕤 等 ISBN：978-7-111-26832-1 定价：33.00元

推荐阅读


用于物联网的Arduino项目开发
实用案例解析


O'REILLY
物联网设备安全
Abusing the Internet of Things
Nitesh Dhanjani 著
林林 陈煜 龚焱哲 译
缪伦 审校


从M2M到物联网
架构、技术及应用
From Machine-to-Machine to the Internet of Things

封面设计　锡彬

CoAP是受限制的应用协议（Constrained Application Protocol）的简称。随着近几年物联网技术的快速发展，越来越多的设备需要接入互联网。虽然对人们而言，连接互联网方便快捷，但是对于那些低功耗受限制设备，接入互联网却非常困难。在当前由PC机和智能手机组成的互联网世界中，信息交换一般通过TCP和HTTP协议实现。但是低功耗受限制设备要实现TCP和HTTP协议也许是一个非常苛刻的要求。为了让低功耗受限制设可以流畅接入互联网，CoAP应运而生。CoAP是一种物联网应用层协议，它运行于UDP协议之上，而不是像HTTP那样运行于TCP之上。CoAP借鉴了HTTP协议大量的成功经验，CoAP和HTTP都使用请求响应工作模式。与HTTP采用文本首部不同，CoAP采用完全的二进制首部，这使得CoAP的首部更短，传输效率更高。CoAP为低功耗受限制设备而生，一个内存仅有20KB的单片机也可以实现CoAP服务器或客户端。

本书主要内容包括：

- 学习CoAP必要的网络基础知识
- CoAP与MQTT、HTTP之间的区别与联系
- CoAP核心内容：二进制首部、工作模式、重传机制、响应码、选项和媒体类型等
- CoAP扩展内容：CoAP资源描述和CoAP观察者
- 使用C语言、Python或Node.js实现CoAP客户端与服务器
- 使用Copper插件和Wireshark调试CoAP

上架指导：计算机/物联网
ISBN 978-7-111-57780-5
9 787111 577805 >
定价：59.00元

投稿热线：(010) 88379604
客服热线：(010) 88379426 88361066
购书热线：(010) 68326294 88379649 68995259
华章网站：www.hzbook.com
网上购书：www.china-pub.com
数字阅读：www.hzmedia.com.cn